Work Report for IoT Device Development for Region Specific Smart Agriculture

Project Number - HCP0057

Principle Investigator: Dr. Babankumar S. Bansod

Report Submitted by
Swastik Bimal Bhattacharya

CSIR - Central Scientific Instruments Organisation
Sector 30C
Chandigarh - 160030, India
July 30, 2024

Contents

Abstract

This project focuses on developing Internet of Things (IoT) devices to enhance smart agriculture practices. By integrating sensors, wireless communication, and data analytics, the project aims to provide real-time monitoring and management of agricultural environments. These IoT devices collect critical data on soil moisture, temperature, humidity, and crop health, transmitting this information to a centralized system for analysis. The system then utilizes this data to optimize irrigation, fertilization, and pest control, ultimately improving crop yields and resource efficiency. By leveraging IoT technology, the project seeks to advance precision agriculture, reduce waste, and promote sustainable farming practices. As part of this project, the following tasks have been undertaken:

- **Development of software for IoT node:** I have developed the software code for the IoT node that can continuously capture data from different sensors attached to it, store it, and transmit it to an IoT gateway for further transmission to a cloud server.
- **Development of software for IoT gateway:** I have developed the software code for the IoT gateway that can continuously receive and handle data from multiple nodes, process and store them, display them, and transmit them to a cloud server.
- **Documentation of Fluoride Sensing System:** I have coordinated with the team in our research group who have been developing the Fluoride Sensing System. Documentation included thorough literature review, electronics design, echanical chassis design, software development, and experiment documentation. Such a sensing system is essential as it can be further extended to measure the fluoride content in soil.
- **Chassis Design for Krishi-IoT device:** Apart from development of the software for the IoT devices, I also took the initiative to design the chassis for packaging both the node and the gateway devices.
- **Development of indigeneous datalogger:** I initiated and spearheaded collaboration with Mindgrove Technologies Pvt. Ltd to co-develop an indigenous IoT datalogger based on Shakti microcontroller developed by IIT-Madras.
- **Procurement of Sensors:** I initiated contacts with different vendors to provide high quality soil, leaf and meteorological sensors for the purpose of this project. Before this, the features of different sensors and their sensing mechanisms have also been thoroughly vetted.
- **Coordination for renovation of laboratory space:** I have spearheaded the coordination efforts to procure furniture and designing the layout for the laboratory space meant for developing IoT devices for agriculture and water quality.
- **Coordination with Administration:** I have coordinated with the administration towards pushing our required purchasing requirements. This included timely submission of bills to the Stores and Purchases section, regularly taking updates from the Accounts section, and regularly addressing the queries raised by these sections for timely payments to the vendors for keeping a longstanding and trustworthy relationship to have access to high-quality materials useful for this project.

Acknowledgements

I would like to express my sincere gratitude to all those who contributed to my work during my tenure under this project. Firstly, I extend my heartfelt thanks to the Principal Investigator of the project, Dr. Babankumar S. Bansod, for his invaluable guidance, insightful feedback, and continuous support throughout this endeavor. His expertise and encouragement was instrumental in shaping the direction and outcome of this work.

I also wish to thank the Council for Scientific and Industrial Research (CSIR) for their financial support, which made this project possible. My appreciation goes to the Director, CSIR-Central Scientific Instruments Organisation for providing the necessary resources and facilities for conducting our research.

Special thanks are due to my colleagues and team members, whose collaboration, dedication, and hard work were crucial to achieving the project goals. Their commitment to excellence and willingness to go the extra mile ensured the high quality of the results.

Lastly, I am grateful to my family and friends for their unwavering support and understanding throughout this project. Their patience and encouragement provided me with the strength and motivation to see the assigned tasks under this project through to its completion.

1 | Introduction

India is predominantly an agrarian nation, with over 50% of the people working in the agricultural sector [11]. This constitutes the primary sector of our economy, along with dairy farming and animal husbandry. While the tertiary sector accounts for the majority of India's GDP, the primary sector contributes only 20%, despite it providing the majority of employment to the current population involved in economic activities. However, foodgrain output surged from 51 million tonnes (MT) in 1950-51 to 250 million MT in 2011-12 [39]. India is one of the top three producers of a variety of goods, including tobacco leaves, rice, wheat, pulses, groundnuts, rapeseeds, fruits, vegetables, sugarcane, tea, jute, cotton, and other commodities[16]. This has been made feasible via targeted interventions through the research, development, and dissemination of solutions and technology to boost agricultural productivity per unit of land. These include the application of pesticides and fertilisers, that have increased crop productivity. This has been an effect of efforts under the Green Revolution to encourage the use of High Yielding Variety (HYV) seeds along with proper use of pesticides, fertilizers, herbicides, etc [28]. This has led to the emergence of a fresh problem of its indiscriminate use [27].

Pollution of water and soil is exacerbated as a result of fertilisers and pesticides used excessively [42]. To appropriately monitor and control the soil, crop, and water health required for a healthy crop yield, focused and strategic interventions are therefore required [31]. Precision agriculture is based on this requirement to employ the appropriate amount of resources at the appropriate time [19]. This is followed by monitoring factor contributing to crop health above and below the canopy, along with soil health. Soil pH, soil nutrients, leaf wetness, soil temperature, soil moisture, incident photosynthetically active radiation, etc. are some of the critical factors that might affect crop output. To advise farmers on the best practices to employ for both improved yield and economic uplifting time-to-time, it is crucial that these factors be continuously monitored [59]. The advisories can consist of the crops that could be planted in accordance with the properties of the soil and the rotation of crops that may be planted in order to preserve the soil's fertility and nutrient content. In addition, this can assist stakeholders and policymakers in making decisions about when and what kind of fertiliser and supplement to apply in order to maintain the soil's potency. This can also assist farmers in implementing good agricultural practices, such cutting back on excessive water consumption or fertiliser use when not needed [17].

Excessive use of fertilizers and pesticides pose a serious risk to public health when water runoff contaminates the land and water supplies. This can lead to the soil losing its potency as well [42]. Therefore, to monitor crop health, soil health, and climatic parameters, CSIO is currently developing an Internet of Things (IoT) enabled datalogger. This datalogger will be able to send data to a cloud database over an IoT network. Subsequently, in addition to monitoring, the same datalogger node might be utilised to control the amount of water and fertiliser applied to the soil and crop, respectively. Constant datalogging requires standardised and dependable sensors. This presents a challenge - a number of sensors and the datalogger's hardware components need to be imported from overseas. This raises the price of development and manufacturing, which is passed on to the farmer who would want to use this type of system. It becomes essential to create an ecosystem that can investigate, create, and use digitally enabled, intelligent solutions that can be inexpensive for farmers [58]. To develop dataloggers, sensors, controlled agriculture environments, soil health maps, crop yield models, variable-rate fertiliser sprayers, and other intervention strategies, CSIO and a consortium of eight CSIR laboratories are working together as part of CSIR's Agri-mission. These technologies are currently being tested on crops like paddy, gerbera, saffron, apple and mint. Different CSIR laboratories are responsible for different work packages under this mission. These include the development of a soil health map by CSIR-4PI along with the development of yield models for different crops. Under this project, CSIR-CMERI has developed a variable-rate fertilizer sprayer, CSIR-CEERI is developing a controlled environment agriculture (CEA) for saffron. CSIR-CSMCRI is developing sensors for detecting Zinc (Zn), Iron (Fe), and Copper (Cu).

At CSIO, current efforts include creating an IoT datalogger with an ESP32 processor that can transmit data over wireless medium using WiFi, LTE, and LoRa to connect to an IoT gateway and cloud. The development is also concentrated on using comparable technology through the Shakti microprocessor-based system, supporting the Government of India's Shakti ecosystem as well as the CSIR's agri-mission. Using these dataloggers as a foundation, data-driven models and decision support systems can be created and integrated with the technology to offer a quantitative and automated solution to various aspects of precision agriculture. These include weather, soil health, and crop yield advisories, along with the

necessary interventions.

With the task assigned to CSIO under the CSIR Agri-mission, following objectives have been currently fulfilled:

- Ideation of an IoT network architecture.
- Development of the software for an IoT node.
- Development of the software for an IoT gateway.
- Experimenting data-transmission to cloud from IoT node and gateway.

This report is organized into 6 chapters. Chapter 1 is the introduction that presents the motivation and objectives of the project to the reader. Chapter 2 discusses the ideation of the Agri-IoT system. It presents the concepts of IoT nodes and gateways, and how the data should be routed using their requisite protocols over the wireless channel. Chapter 3 describes the basic features of the IoT node and its software development. It presents the readers with the different protocols using which the node communicates with the sensor, the data format of the measurement from the sensors, and how the data is being transmitted to the gateway. The packaging of the node hardware is also discussed. Chapter 4 deals with the development of the IoT gateway. It touches upon some concepts of data structures and real-time embedded systems. The functionalities and the software development for the gateway is elucidated upon. The packaging of the gateway is also looked into. Chapter 5 is about the demonstrations and experimentation of the IoT node and gateway setup under laboratory and field conditions. Chapter 6 tells the reader about the future scope of work and how can the development go ahead from this point.

2 | Agri-IoT Ideation

Water scarcity, erratic weather patterns, and pest infestations are a few of the many difficulties Indian farmers have been traditionally facing [21]. These difficulties have a big influence on their livelihood and output. Due to the efforts under the Green Revolution, HYV seeds, fertilizers, herbicides, pesticides have been introduced and the farmers have been proactively utilizing these resources, thus alleviating some of the problems [43]. However, newer methods come with newer problems, which include contamination of water and soil. The integrity and potency of soil is also dependent on the practices used by the farmers. A potential answer to these issues is the application of the Internet of Things (IoT) to agriculture. Farmers in various parts of India can optimise irrigation and preserve water by using real-time data on soil moisture levels from IoT devices [58][55]. Weather-monitoring sensors assist farmers in making well-informed decisions regarding when to sow and harvest, reducing the hazards associated with erratic weather patterns. IoT-powered automated machinery may more precisely and efficiently carry out operations like planting, fertilising, and harvesting, alleviating labour shortages and increasing production. By minimising crop loss and lowering the need for chemical treatments, early detection and reaction to pest infestations via IoT promote sustainable farming practices [25]. IoT data also makes it easier for farmers to monitor crop progress and forecast harvests, which helps them plan better for the market and get fair prices for their produce. IoT minimises post-harvest losses by streamlining supply chain logistics to guarantee that produce reaches markets and consumers in the freshest possible condition [56]. Large-scale data collection and analysis facilitates the discovery of trends and patterns, which enhances agricultural techniques and promotes sustainability over the long run [58].

With this premise, this chapter discusses IoT networks and how they could be used in smart agriculture. This chapter also discusses some important aspects of IoT networks, which include IoT devices and communication paradigms.

2.1 | IoT Networks

The seamless connection and communication between diverse equipment and systems made possible by IoT networks is revolutionising smart agriculture by providing real-time monitoring and management of agricultural activities [58]. Through the collection, transmission, and analysis of data from sensors and smart devices, these networks enable farmers to maximise their operations and make well-informed decisions [32]. The application of IoT nodes and gateways, which are essential for data transmission and gathering, is a critical component of IoT networks in agriculture [41][30].

The endpoints in an IoT network that gather environmental data are known as IoT nodes. These nodes in smart agriculture could be gadgets such as automated irrigation systems, GPS-enabled tractors, weather stations, and soil moisture monitors [36]. Farmers can optimise irrigation and preserve water resources with the help of soil moisture sensors, which offer vital information on water levels [55]. Farmers can more precisely plan planting and harvesting periods thanks to weather stations that are outfitted with Internet of Things (IoT) sensors that measure temperature, humidity, and other meteorological parameters [57]. Precision farming is made easier by GPS-enabled tractors and drones, which automate repetitive chores like pest management, fertilisation, and planting while maintaining accuracy and efficiency.

IoT nodes are connected to centralised data storage systems or the cloud via gateways, which act as intermediaries. They collect, process, and send data to the cloud for additional analysis and decision-making from several IoT nodes. Gateways are essential to the management and effective use of data gathered from diverse sensors and devices in smart agriculture. Gateways provide for smooth data flow and real-time agricultural process monitoring by bridging the gap between IoT nodes and the cloud [41].

The foundation for IoT networks in smart and precision agriculture has been established by [5], Offering a global solution to this problem that provides a consolidated and integrated IoT system where data collection, monitoring , control and communication platform are managed using a single platform. Based on the framework suggested in [5], there have been cognitive-based decision support system for agriculture that has been developed by [8] that uses queries from the client to fetch real-time data from IoT sensor networks and provides data-driven insights to make any agronomical recommendation. There have also been attempts at creating a software defined network that use fly-by drones, as claimed by [52]. There has also been development into creating an IoT-based controlled agriculture environment as claimed by [53].

Keeping these prior arts as a baseline, an IoT network for smart agriculture is proposed here with emphasis on the following major components:

- **IoT nodes:** These are the endpoints of the IoT network that are connected to different sensors, namely meteorological, soil and leaf parameter sensors [14]. Data from these sensors are recorded by the nodes using protocols such as UART, I2C, RS485, 4-20mA, etc [46]. Apart from capturing data from the sensors, the nodes can locally store the data with timestamps based on Navigation data using GNSS modules [23]. The measurements are then transmitted to the gateway device over the wireless channel. The timekeeping is can be done using GNSS module or from an on-board Real-Time Clock (RTC) module or circuitry. The heart of the node hardware is an ESP32 System-on-Chip (SoC) [50]. The development of the software to perform the requisite operations is done using the Arduino framework [7]. The hardware has circuits of battery protection, battery charging, battery level gauge, RS485 MODBUS serial communication, SD Card interfacing, 4-20mA sensor interfacing, WiFi and LoRa.

- **IoT gateways:** As mentioned in the earlier, gateways are intermediaries between the node and a data collection/processing system [20][44]. The gateway consolidates the data from different nodes. The data is segregated based on the site and they could be seen on an LCD display. The gateway receives data over the wireless medium. The collected data is then sent to the data processing centre over the wireless channel. At the heart of the gateway hardware is Raspberry Pi 4B [54]. The software uses a multithreading approach into performing the operations of data reception, data processing, data transmission and GUI updates [10][40].

- **Communication modes:** The data transmission is done over the wireless channel using primarily two protocols: WiFi [13] and LoRa [47]. High-speed data transfer over short to medium distances is made possible by WiFi, which makes it perfect for applications like farm management systems and greenhouses that demand enormous data volumes and real-time monitoring [38]. However, low-power, long-distance communication is an area where LoRa (Long Range) technology shines, which makes it ideal for sending data from remote sensors dispersed throughout large agricultural fields [34]. In large farming areas, LoRa's ability to provide dependable connectivity over long distances while consuming little power is especially helpful for ongoing crop health, weather, and soil moisture monitoring [34]. By combining these technologies, an IoT network that is both comprehensive and versatile can be created to satisfy the various demands of contemporary agriculture [33][12].

- **Network architecture:** The communication modes, nodes and gateways form the complete network architecture. The sensors are connected to the node hardware through physical connections following a certain protocol (such as RS485 and 4-20mA). In the case of RS485, the node sends instruction to the sensor, which in turn replies with the value stored by the register. In case of 4-20mA, the observable will be recorded proportionately based on the current produced by the sensing mechanism. These measurements are captured, processed by the node, and then stored locally with geo-location and timestamping. After the data is processed, the information is transmitted over the wireless channel to the gateway. The gateway receives the measurements and processes it based on the node location. The gateway then has a UI that gets updated, and then using wireless medium the data is transmitted to the cloud. Figure 2.1 shows the proposed IoT network architecture.

2.2 | IoT Nodes and Gateways

In this proposed IoT network for smart agriculture, the devices used could be categorized as one at the end-point, and one in between the final data center and the endpoint. They are called IoT Nodes and IoT Gateways respectively. Nodes collect and sometimes process data from various sources, while gateways aggregate, process, and transmit this data for further analysis and action.

2.2.1 | Nodes

The IoT nodes are based on ESP32 SoC [50] having the following features:

- **Power Supply:** The node can be powered using main power supply with the help of a Switched Mode Power Supply (SMPS). The node needs a supply of 5V to power up.

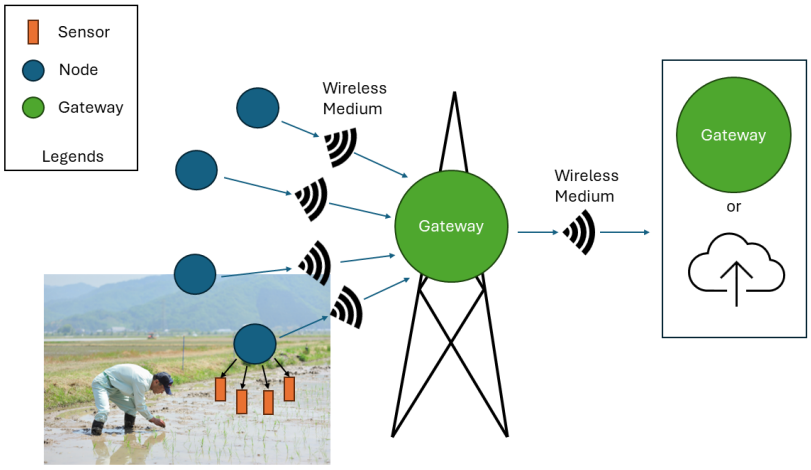

Figure 2.1: Proposed IoT network architecture

- **Power backup:** The node can also have a power-backup using batteries upto 5V. In the presence of main power supply, the node switches to mains supply and can charge the battery with a battery charging and protection circuit. There is also a battery level gauge that provides the battery state to the ESP32 SoC. In the absence of mains power supply, the node switches to operate using battery power for a limited amount of time.
- **Sensor Interfaces:** The node has the capability to directly interface with RS-485 sensors and two 4-20mA sensors. There are specialised circuits that help the sensors interface with the SoC. Some of the sensors that could be interfaced are Soil Moisture, Soil Electrical Conductivity (EC), Soil Temperature, Meteorological sensors and leaf sensors.
- **Local Storage:** There is a provision for SD card storage to save the measurements into a microSD card. The module for microSD card is interfaced with the SoC using SPI protocol.
- **Timekeeping and geolocation:** There is an on-board RTC circuit for local timekeeping in the absence of a GNSS module. The GNSS module provides geolocation and timing services using the measurements using different navigation satellite systems such as the Global Positioning System (GPS).
- **Wireless Transmission:** The ESP32 SoC [50] used in the node has in-built WiFi module, which enables it to communicate with the gateway over the WiFi mode. To facilitate transmission over longer distances, there is also a circuit interfacing RFM95 module for LoRa to the ESP32 SoC using SPI protocol. In India, LoRa transmission in outdoors is permitted if the 868MHz frequency band is used.

2.2.2 | Gateways

IoT gateways act as intermediaries between IoT nodes and the cloud or centralized data storage systems. They aggregate data from multiple nodes, perform initial processing, and then transmit this data to the cloud for further analysis and storage. The IoT Gateways in this project os based on a Raspberry Pi 4B with the following features:

- **Multicore Processing:** The Raspberry Pi 4B has a 4-core microcontroller [54] that provides the feature of parallelism to run multiple tasks concurrently.
- **Wireless transmission:** The gateway has the capability to acquire data from the node with the help of in-built WiFi interface using the Message Queue Telemetry Transport (MQTT) protocol [26]. It also has the feature of communication using LoRa [47] and LTE [6], the particulars of which will be discussed in Chapter 4.
- **Timing and Geolocation:** The gateway has an added module for communication using LTE [6][4], which also includes facility of GNSS location and timing services. This is used for setting the time of the gateway in the network time, along with capturing the initial location of the gateway.

- **Data Processing:** The data from different nodes are captured and processed by the gateway, and are segregated based on the node locations and the day of year in which the measurements have been recorded.
- **User Interface:** A UI has also been developed for a touch-screen LED display to show the user incoming measurements from different sites in real-time.

2.3 | Communication in IoT Networks

A network is an assembly of devices that are connected to one another and communicate with one another to exchange data and resources [51][18]. These gadgets, which are also called nodes, might be servers, PCs, sensors, and other hardware. Networks come in many sizes, ranging from a tiny local area network (LAN) to a large global network such as the Internet, and they enable the exchange of data using different communication protocols. The seven layers of the Open Systems Interconnection (OSI) architecture serve as a conceptual foundation for comprehending and implementing network protocols [51][18]. Every layer interacts with the layers immediately above and below it and serves a specific purpose. Through the standardisation of network services, the OSI model ensures compatibility across various hardware and software systems. The OSI model's seven layers are:

- **Physical Layer:** Manages the physical connection between devices and the transmission of raw binary data.
- **Data Link Layer:** Ensures reliable data transfer between two physically connected devices, managing errors and flow control.
- **Network Layer:** Determines the best physical path for data to reach its destination, handling routing and addressing.
- **Transport Layer:** Provides end-to-end communication, ensuring complete data transfer with error checking and recovery.
- **Session Layer:** Manages sessions or connections between applications, handling setup, maintenance, and termination.
- **Presentation Layer:** Translates data between the application layer and the network, handling encryption, compression, and translation.
- **Application Layer:** Interfaces directly with end-user applications, providing network services to applications.

The OSI model is essential to comprehending how various devices interact and communicate in Internet of Things networks [35]. IoT networks are made up of different actuators, sensors, and other intelligent devices that exchange and gather data in order to carry out particular activities. Developers and engineers may make sure that various devices, independent of their manufacturers or particular implementations, operate together flawlessly by using the OSI model [45].

2.3.1 | Physical and Data Link Layer

The OSI model's Physical and Data Link layers are in charge of the hardware and protocols that make it possible for raw data to be transmitted across a variety of communication channels. These layers cover wireless technologies such as WiFi, and LoRa in smart agriculture. Vast farms may now install a vast number of sensors and other equipment thanks to these technologies, which also provide the infrastructure needed for ongoing data collection and monitoring.
In the outdoor environment of an agricultural field makes the use of wireless medium an attractive option for the communication between the nodes and gateways over wireless channels an attractive option. However, one needs to note that wireless channel is always a shared medium. So, a few things need to be considered when using the wireless channels:

1. **Multiple Access Control [24]:** Within the Data Link layer, multiple access control (MAC) plays a crucial role in controlling how different devices share a single communication medium. Effective MAC protocols are crucial for preventing collisions and ensuring efficient data transfer in smart agriculture, where multiple IoT devices and sensors may be transmitting data at the same time.

Different kinds of MAC protocols, including Time Division Multiple Access (TDMA) [22], Carrier Sense Multiple Access (CSMA) [18], and Code Division Multiple Access (CDMA) [18], can be distinguished. For instance, TDMA lowers the possibility of collisions by assigning distinct time periods for data transmission to every device. In agricultural situations, where several sensors and devices must transmit data without interference, this is vital. The proposed network uses CSMA/Collision Avoidance (CSMA/CA), which is discussed in Chapter 3.

2. **Communication over Wireless Channel:** Over the wireless channel, communication can be done using WiFi or LoRa with the help of following protocols:
   - **MQTT (Message Queuing Telemetry Transport) using WiFi (IEEE 802.11):** MQTT [26] is a lightweight, publish-subscribe messaging protocol designed for constrained devices and low-bandwidth, high-latency networks. In smart agriculture, nodes equipped with various sensors can use MQTT to send collected data to gateways. The protocol ensures efficient data transmission with minimal overhead, making it ideal for scenarios where power and bandwidth are limited. Now, this is a protocol that operates on the application layer. However, it uses the WiFi [13] IEEE 802.11 protocol for multiple access control for the data link layer.
   - **LoRa (Long Range):** LoRa is a low-power wide-area network (LPWAN) protocol that enables long-range communication between IoT devices. Nodes using LoRa can transmit data over several kilometers to a gateway. This capability is particularly beneficial in large agricultural fields where deploying wired connections or short-range wireless networks would be impractical.

2.3.2 | Network Layer

Data packet routing throughout the Internet of Things is controlled by the network layer. This is the process of sending data from field sensors to gateways and then cloud servers in smart agriculture. This layer is necessary for the network to be scalable, enabling it to manage several devices dispersed over large agricultural regions. By improving connectivity and lowering latency, optimised routing techniques make it possible to gather and analyse data from every area of a farm effectively. The gateways form the part of the network layer as they act as a bridge between the network between the nodes and gateways, and the network connection between the cloud or data collection centre.

2.3.3 | Transport Layer

Integrity and dependability of end-to-end communication are guaranteed by the transport layer. To control data flow and guarantee complete data transfer, protocols like TCP/IP and UDP are employed. This layer in smart agriculture facilitates the safe transfer of vital information from sensors to centralised processing systems, such as soil moisture content and meteorological conditions. It manages retransmissions in the event of data loss, guaranteeing that no important information is overlooked—a critical aspect of timely decisions in agricultural operations.

2.3.4 | Session Layer

The Session layer establishes, manages, and terminates communication sessions between devices. In an agricultural IoT network, this layer can be used to synchronize data exchanges between field sensors and control systems. For example, a session could be established for periodic data uploads from a weather station to a central database, ensuring that the data transfer is coordinated and completed without interruptions. This layer supports the seamless operation of automated systems, such as irrigation controls that rely on continuous data streams. The gateway uses Hyper Text Transfer Protocol (HTTP) [9] to transfer the collected data from the nodes to the cloud (currently ThingSpeak [37]). A session is opened between the gateway where the device gets assigned an IP address and a port for a dedicated connection with the ThingSpeak server.

2.3.5 | Presentation Layer

The Presentation layer is in charge of encrypting, compressing, and translating data so that the application layer can comprehend data from diverse sensors. This is transforming unprocessed sensor data into standardised formats for analysis in smart agriculture. Protecting sensitive agricultural information requires maintaining data integrity and security throughout the communication process, which is what this layer makes sure of.

2.3.6 | Application Layer

Through direct interfaces with end-user programmes, the application layer offers network communication services. This layer in smart agriculture offers a range of applications, including automated control systems, smartphone apps for real-time monitoring, and farm management systems. It helps farmers make decisions by turning the data that has been gathered and processed into insights that can be put into practice.

2.4 | Proposed Network Architecture for Agri-IoT

An enhanced Internet of Things network is suggested to optimise data collecting and analysis in smart agriculture. It incorporates a range of sensors and communication protocols to improve scalability, dependability, and efficiency.

- **Sensor Nodes:** The sensor nodes in this network are built around the ESP32 System on Chip (SoC). The ESP32 is chosen for its versatility, low power consumption, and integrated WiFi and Bluetooth capabilities. Each sensor node is equipped with the following features:
  - **RS485 Communication:** RS485 is a robust, long-distance communication protocol often used in industrial environments. It connects to sensors that monitor various agricultural parameters such as soil moisture, temperature, and humidity.
  - **4-20mA Sensors:** These sensors are widely used in industrial applications for their accuracy and resistance to electrical noise. They measure parameters like soil pH and nutrient levels.
  - **Timekeeping Services:** Each node has an integrated Real-Time Clock (RTC) to timestamp the collected data, ensuring precise time synchronization across the network.
  - **Geolocation Services:** GPS modules are included in the nodes to provide geolocation data, allowing for precise mapping and monitoring of specific areas within the farm.
- **Data Transmission:** The sensor nodes transmit collected data to a central gateway using MQTT (Message Queuing Telemetry Transport) [26] over WiFi [13] or LoRa [47] at 868MHz.
  - **MQTT over WiFi:** For areas with possible reliable WiFi coverage, MQTT over WiFi is used due to its low latency and high data throughput, ensuring real-time data transmission.
  - **LoRa at 868MHz:** In remote or large agricultural areas where WiFi is not feasible, LoRa provides a long-range, low-power communication solution. LoRa's 868MHz frequency band is suitable for overcoming physical obstructions and covering extensive farm areas.
- **Gateway:** The central gateway in this network is based on the Raspberry Pi 4B, a powerful and flexible computing platform.
  - **Data Reception:** The gateway receives data from sensor nodes using MQTT over WiFi or LoRa. It is equipped with both WiFi and LoRa modules to ensure compatibility with different transmission methods.
  - **Data Processing:** Upon receiving data, the gateway processes it to create structured datasets. This involves organizing the data by node site, timestamp, and geolocation, making it ready for analysis and visualization.
  - **User Interface (UI):** The gateway hosts a local UI that displays real-time data and analytics for on-site monitoring. Farmers can access this UI via any device connected to the local network, providing immediate insights into farm conditions.
- **Data Transmission to the Cloud:** After processing, the structured data is transmitted to the Thingspeak cloud platform using HTTP over a wireless channel.

3 | Node Development

An IoT node is an essential part of the IoT ecosystem, serving as a link between the digital and physical worlds [41]. Typically, these nodes in smart agriculture consist of sensors, actuators, and communication interfaces that gather information on temperature, humidity, soil moisture content, and other vital characteristics [38]. After processing, the data is sent to a central computer or other devices for analysis, allowing farmers to make well-informed decisions about pest management, fertilisation, and irrigation. IoT nodes are useful in smart agriculture because they can give real-time data and control, which improves productivity, judgement, and resource management.

This chapter explores the creation of IoT nodes designed for smart agriculture, including an overview of their hardware characteristics, function in the Physical and Data Link layers of the OSI model [51][18], and software development with the Arduino Integrated Development Environment (IDE) [7]. With an emphasis on the software development with sensors required in agricultural applications, it will start by examining the fundamental parts of an Internet of Things node, such as microcontrollers, sensors, communication modules, and power supplies. The importance of IoT nodes in the Physical and Data Link layers will be covered in this chapter, along with how they handle data transfer, and provide dependable connection in expansive and remote farming areas. This chapter will also walk readers through the Arduino IDE programming procedure for IoT nodes.

3.1 | Hardware Features of the Node

Based on the ESP32 WROOM32UE [50], the hardware architecture of the IoT node for smart agriculture includes a number of essential features to guarantee dependable, versatile, and strong performance in a range of agricultural situations. The essential hardware parts and features built into the Internet of Things node are described in detail in this section. The following major features have been integrated into the Printed Circuit Board (PCB) of the node:

- **Power Input Switching:** To guarantee continuous functioning, the IoT node is equipped with a variety of power input options. Battery power, an SMPS, or a USB port can all be used to power it. The node automatically switches to battery power to ensure ongoing operation when the USB or SMPS, which serve as the primary power source, are unavailable.
- **Battery Charging and Protection:** When the primary power supply is connected, the battery attached to the IoT node can receive up to 5V of charging. In order to prevent short circuits, overcharging, and overdischarging, the node has battery protection functions. A battery level gauge also keeps an eye on the condition and health of the battery, giving current information on the voltage that is available.
- **Voltage Regulation:** The node's power supply system includes voltage regulators that provide 3.3V and 5V supply points. This ensures that both the SoC and peripheral components receive the appropriate voltage levels. The voltage regulation for the SoC and peripherals is isolated to prevent interference and ensure stable operation.
- **File Storage:** An SD card module interface is included for file storage, enabling the node to store large amounts of data locally. This feature is crucial for logging sensor data and other information that can be accessed or analyzed later.
- **LoRa Communication:** The node is equipped with an onboard LoRa module that operates on the 868MHz frequency band, as allowed in India. This enables long-range, low-power communication, making it ideal for transmitting data over large agricultural fields.
- **Real-Time Clock (RTC):** An onboard real-time clock (RTC) provides accurate timekeeping, essential for timestamping data and scheduling tasks. This ensures that data collected from the sensors is accurately recorded and can be correlated with specific times and events.
- **Connectivity Options:** The node features onboard circuitry for WiFi connectivity, allowing for easy integration into local networks. Additionally, it supports GSM and GNSS (Global Navigation Satellite System) for 2G connectivity and precise Position, Navigation, and Timing (PNT) services through UART using GPS L1. There are plans to integrate NAVIC L5. This ensures that the node can function as a gateway with internet connectivity and accurate geolocation capabilities.

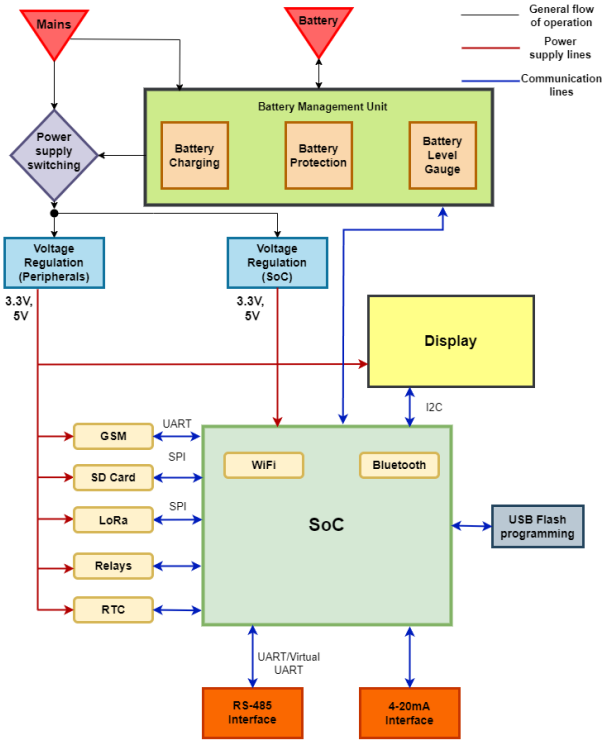

Figure 3.1: Operations and facilities in the node hardware.

- **Human-Machine Interface (HMI) Display:** An HMI display is included to view sensor data and interact with the local IoT network. This user interface is essential for standalone operation, providing real-time information and control at the site. Currently, it supports only a simple OLED display to show the measurements captured in real-time.
- **Programming:** A UART-based programming port is provided for convenient programming and debugging of the node. This feature simplifies the development process and allows for easy updates and modifications to the node's firmware.
- **RS485 interface:** RS485 interfaces are provided to connect sensors with the node hardware through an RS485 to UART/software serial interfacing circuit.
- **4-20mA interface:** 4-20mA sensor interfaces are provided to connect sensors with the node hardware a circuit where the voltage drop across a resistance is measured by the ESP32.

The reader can review the Appendix A for more information into the components used and the schematic circuits of the node hardware.Figure 3.1 shows the overall blocks that are present in the node hardware along with the flow of operations. Figure 3.2 shows the top and bottom sides of the node hardware PCB.

3.2 | Physical and Data Link Layer: Communication between Sensors and the Node

The actual raw data transmission and reception between devices in an IoT node is handled by the physical layer. Within the framework of smart agriculture, the ESP32 SoC interfaces with several sensors through the use of the RS-485 MODBUS protocol, which is a reliable and extensively utilised industrial standard. This protocol ensures dependable data transfer over large farm areas and is especially well-suited for long-distance communication in electrically loud conditions.

Sensor data can be sent to the ESP32 SoC via a differential serial communication connection using the RS-485 MODBUS [49] protocol. Because it can accommodate several devices on the same bus, RS-485 is the recommended protocol for gathering data from a network of sensors dispersed over a broad field. With software serial capabilities, the ESP32 SoC [50] establishes a virtual serial port to communicate with the RS-485 bus. With this configuration, the ESP32 can process sensor data and take appropriate action based on predetermined criteria.

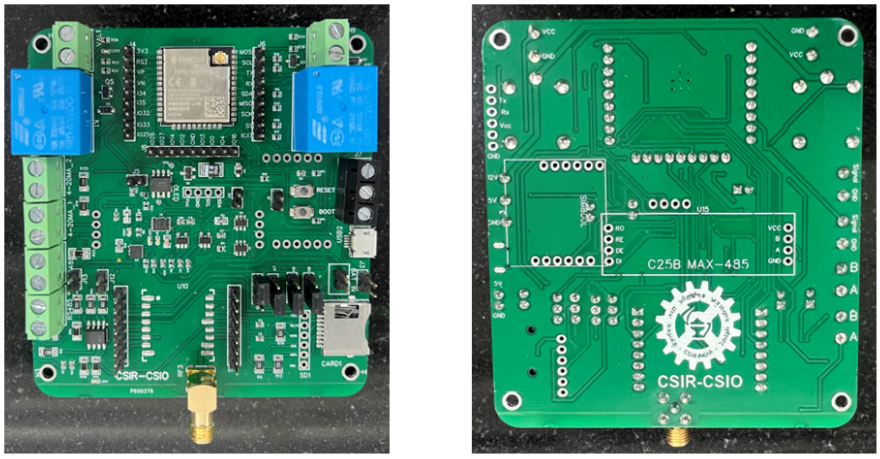

Figure 3.2: Top and bottom sides of the node PCB hardware.

To interface RS-485 with the UART (Universal Asynchronous Receiver/Transmitter) of the ESP32 SoC, specific circuit components are required:

- **RS485 Transceiver:** The MAX485 [2] is a commonly used transceiver that converts RS-485 signals to TTL-level signals compatible with the UART of the ESP32. It supports half-duplex communication, allowing the same pair of wires to be used for both transmitting and receiving data.
- **Termination Resistors:** To prevent signal reflections on the bus, termination resistors (typically 120 ohms) are placed at both ends of the RS-485 communication line. These resistors match the impedance of the communication cable, ensuring signal integrity.
- **Pull-up and Pull-down Resistors:** These resistors ensure a known state for the data lines when the bus is idle, improving noise immunity. Pull-up and pull-down resistors help maintain a defined logic level, preventing false triggering.
- **Power Supply Decoupling:** Capacitors are used to decouple the power supply, providing stable voltage to the transceiver and reducing noise in the power lines.

In the node's circuit, the MAX485 transceiver connects to the RS-485 bus on one side and to the ESP32's UART pins on the other. The DE (Driver Enable) and RE (Receiver Enable) pins of the MAX485 are connected to the GPIO pins of the ESP32, allowing the microcontroller to control the direction of data flow. In this case, the circuit to interface UART to RS-485 is on-board, with the DE and RE pins shorted. When transmitting data, the DE pin is set high, and since the RE pin is active low for receiver operation, it does not receive data when set high. Conversely, for receiving data, the DE pin is set low, and the since the RE pin is active low for receiver operation, it does not transmit but it receives data when set low. This setup allows the ESP32 to communicate efficiently with multiple sensors over the RS-485 bus. The software serial port created on the ESP32 facilitates this communication, enabling the microcontroller to read sensor data, process it, and perform necessary actions or transmit the data to a central gateway or cloud server for further analysis.
The datalogger also has the capability to read measurements from 4-20mA sensors. 4-20mA current loop sensors are widely used in agricultural environments for their accuracy, and since many sensing mechanisms to measure quantities such as soil pH, soil NPK, etc. are based on amperometric principles, where the main observable is current at a characteristic constant voltage. These sensors convert physical measurements into a current signal that ranges from 4 to 20mA, with 4mA typically representing the minimum measurement value and 20mA representing the maximum. The basic characteristic features in the interfacing of 4-20mA sensors and the ESP32 SoC are as follows in the form of on-board circuit:

- **Current-to-Voltage Converter:** To interface 4-20mA sensors with the ESP32, a current-to-voltage converter circuit is necessary. This circuit typically includes an operational amplifier (op-amp) configured as a transimpedance amplifier, which converts the 4-20mA current signal into a corresponding voltage that can be read by the ESP32's analog-to-digital converter (ADC).

- **Precision Resistors:** These resistors are used in the current-to-voltage conversion circuit to ensure accurate conversion of the current signal into a voltage. The resistor value is selected based on the desired output voltage range and the sensor's current range.

The integration of 4-20mA sensors into the IoT node involves connecting the sensors to the current-to-voltage converter circuit, which then feeds the resulting voltage into the ESP32's ADC. This setup allows the ESP32 to read and process the sensor data accurately.

3.3 | Data Processing in the Node

After acquiring data from the sensors with the help of RS-485 and the 4-20mA interfaces, the raw data needs to be processed into information so that it can be stored locally and transmitted to the gateway in a structured manner. The values from different sensors are stored in the form of JavaScript Object Notation (JSON) object. The JSON object has following features:

- **site_id:** Identifier for the site at which the node is installed.
- **lat:** Latitude of the node location in degrees. Positive for northern latitudes and negative for southern latitudes.
- **lon:** Longitude of the node location in degrees. Positive for eastern longitudes and negative of western longitudes.
- **year:** Year of data capture in UTC.
- **doy:** Day of year in the calendar in UTC.
- **hh:** Time hours in UTC.
- **mm:** Time minutes in UTC.
- **ss:** Time seconds in UTC.
- **ph:** The value of Soil pH between 0-14.
- **ph_sensor_address:** Slave ID of the RS-485 soil pH sensor.
- **ph_sensor:** Identifier for the RS-485 soil pH sensor in use.
- **ph_unit:** Unit of measurement of pH. Unitless or pH is the value of unit by default.
- **npk_sensor:** Identifier for the RS-485 soil NPK sensor in use to measure soil Nitrogen, Phosphorous and Potassium.
- **npk_n:** Value of nitrogen content in soil.
- **npk_sensor_address:** Slave ID of the RS-485 soil NPK sensor.
- **npk_unit:** Unit of measurement of Nitrogen, Phophorous and Potassium. mm/kg is the value of unit by default.
- **npk_p:** Value of phosphorous content in soil.
- **npk_k:** Value of potassium content in soil.
- **rain_sensor:** Identifier for the RS-485 tipping bucket rain sensor in use.
- **rain:** Value of rainfall measured.
- **rainfall_sensor_address:** Slave ID of the RS-485 rainfall sensor.
- **rain_unit:** Unit of measurement of rainfall. mm is the value of unit by default.
- **light:** Value of incident sunlight in terms of luminosity.
- **light_sensor_address:** Slave ID of the RS-485 light sensor.
- **light_sensor:** Identifier for the RS-485 photodiode light sensor in use.

- **light_unit:** Unit of measurement of incident light. Lux is the value of unit by default.
- **soil_moisture:** Value of soil moisture at the place of installation of the combined soil moisture, temperature and Electrical Conductivity (EC) sensor.
- **soil_temp:** Value of soil temperature at the place of installation of the combined soil moisture, temperature and EC sensor.
- **soil_ec:** Value of soil EC at the place of installation of the combined soil moisture, temperature and EC sensor.
- **soil_temp_moisture_ec_sensor:** Identifier for the RS-485 combined soil moisture, temperature and EC sensor in use.
- **soil_temp_moisture_ec_sensor_address:** Slave ID of the RS-485 combined soil moisture, temperature and EC sensor in use.
- **soil_moisture_unit:** Unit of measurement of soil moisture. %VWC is the value of unit by default.
- **soil_temp_unit:** Unit of measurement of soil temperature. Celsius is the value of unit by default.
- **soil_ec_unit:** Unit of measurement of soil EC. μS/cmm is the value of unit by default.
- **leaf_sensor:** Identifier for the RS-485 leaf temperature and wetness in use.
- **leaf_sensor_address:** Slave ID of the RS-485 combined leaf temperature and wetness sensor in use.
- **leaf_wetness:** Value of leaf wetness measured.
- **leaf_wetness_unit:** Unit of measurement of incident light. $\%RH$ is the value of unit by default.
- **leaf_temp:** Value of leaf temperature measured.
- **leaf_temp_unit:** Unit of measurement of incident light. Celsius is the value of unit by default.

In the software, this JSON is called **node_doc**. The data, once stored into the above JSON object, is serialized as a string terminated using a newline escape sequence and is then stored locally in a JSON file named as **{site_id}_{doy}.json** Therefore, as the data is captured continuously, the strings are appended to the JSON file for the respective day of year in Universal Time Coordinate (UTC). This is done so that the files could further be processed easily for data analytics insights using a language such as Python that are commonly used in statistical analysis and data science. After the data is stored locally, this JSON is then processed into separate JSON objects for different observables to be sent to the gateway. These JSONs are defined as **gateway_doc_{obs}_{manufacturer}**. Depending on the manufacturer of the sensor, the field **manufacturer** will either be emply by default, or a suffix based on the manufacturer's name. Table 3.1 shows the values of **obs** and their corresponding observable with their MQTT topics. The **gateway_doc_{obs}_{manufacturer}** has the following fields:

- **site_id:** Identifier for the site at which the node is installed.
- **device_id:** Identifier for the RS-485 sensor in use to send the data.
- **lat:** Latitude of the node location in degrees. Positive for northern latitudes and negative for southern latitudes.
- **lon:** Longitude of the node location in degrees. Positive for eastern longitudes and negative of western longitudes.
- **year:** Year of data capture in UTC.
- **doy:** Day of year in the calendar in UTC.
- **time:** Time hours in UTC using **hh:mm:ss** format.
- **Quantity:** Observable being measured.
- **Value:** Value of the observable as measured.

- **Unit:** Unit of measurement for the observable.

As seen from the list of fields of **gateway_doc_{obs}_{manufacturer}**, the value in the field **Quantity** determines the value of **obs** in **gateway_doc_{obs}_{manufacturer}** using 3.1. These JSONs are then published by the MQTT client for their corresponding topics. The MQTT client on the gateway subscribes to these topics, listens, and receives the published JSONs from the nodes.

Table 3.1: Observables and their corresponding suffix (**obs**) and MQTT Topic.

Sr. No.	Observable	Suffix	MQTT Topic
1.	pH	pH	pH_data
2.	Nitrogen	n	n_data
3.	Phosphorous	p	p_data
4.	Potassium	k	k_data
5.	Rainfall	rain	rain_data
6.	Light	light	light_data
7.	Soil Moisture	soil_moist	s_moist_data
8.	Soil Temp	soil_temp	s_temp_data
9.	Soil EC	soil_ec	s_ec_data
10.	Leaf Temp	leaf_temp	l_temp_data
11.	Leaf Wetness	leaf_wet	l_wet_data

3.4 | Application Layer: Data Transmission from the Node

The Application Link Layer of the OSI model plays a crucial role in ensuring reliable data transfer to provide service to the end user. Therefore, it provides a connection between nodes and gateways in an IoT network, where the Gateway is the end user. In the context of smart agriculture, the Message Queuing Telemetry Transport (MQTT) [26] protocol is widely used for its lightweight and efficient communication capabilities. This section explores how MQTT is employed in the application layer to facilitate the transmission of sensor data from the IoT node to the gateway, focusing on its implementation, benefits, and operational specifics.

MQTT is a publish/subscribe messaging protocol designed for low-bandwidth, high-latency, and unreliable networks. It is particularly suited for IoT applications due to its minimal overhead, simplicity, and efficiency in handling large numbers of devices. In an MQTT network, clients (nodes) publish messages to topics, and other clients (gateways or servers) subscribe to these topics to receive the messages.

In an MQTT transmission, the WiFi protocol (IEEE 802.11) is used in the data-link layer as it can handle data originating from multiple sources. A WiFi client is initiated in the ESP32. The node and the gateway uses the Internet Protocol (IP) as a Network Layer protocol to determine the unique address of the devices within the network. To connect to the MQTT broker, Transmission Control Protocol (TCP) is used for a reliable communication through a unique port. Now, there is no inherent Session and Presentation Layer protocols in use with the MQTT workflow. However, in the session layer, CONNECT/DISCONNECT messages for controlling the connectivity with the gateway. The Presentation Layer formats the data to be used by the Application Layer protocol. In the case of this node, the Presentation layer serializes the JSON object after data processing for transmission using MQTT.

As a data-link layer protocol, as mentioned, the node uses WiFi when using MQTT for transmission. One of the key aspects of Wi-Fi at the data-link layer is its mechanism for multiple access control, which is essential for managing how multiple devices share the same wireless medium. The data-link layer in Wi-Fi is divided into two sublayers:

1. **Logical Link Control (LLC) Sublayer:** It provides an interface between the network layer and the MAC sublayer. It is also responsible for multiplexing protocols transmitted over the MAC layer (such as IP) and optionally providing flow control, acknowledgment, and error notification.
2. **Media Access Control (MAC) Sublayer:** It is responsible for coordinating access to the shared wireless medium. It implements mechanisms for multiple access, collision avoidance, and ensuring data integrity.

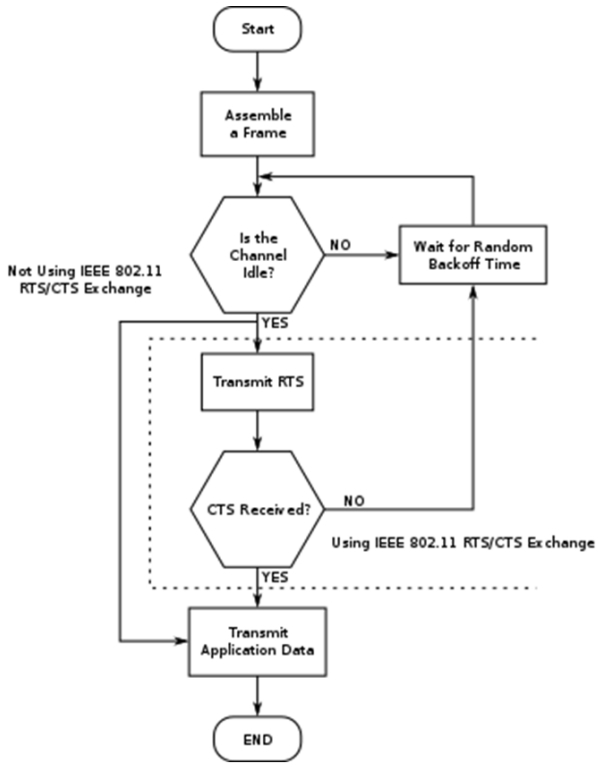

Figure 3.3: CSMA/CA multiple access control protocol flowchart.

The key protocol for multiple access control in Wi-Fi is Carrier Sense Multiple Access with Collision Avoidance (CSMA/CA). Here's how it works (also shown in Figure 3.3):

- **Carrier Sensing:** Before attempting to transmit data, a device listens to the channel to check if it is free (i.e., no other device is transmitting). If the channel is busy, the device waits for a random backoff period before trying again.
- **Collision Avoidance:** Unlike wired networks (which use CSMA/CD), Wi-Fi uses CSMA/CA because collisions cannot be detected easily in wireless networks. To avoid collisions, Wi-Fi devices use a combination of carrier sensing and random backoff periods.
- **Random Backoff:** If the channel is busy, the device waits for a randomly chosen period of time before rechecking the channel. This reduces the probability of multiple devices trying to transmit simultaneously after the channel becomes free.
- **Request to Send / Clear to Send (RTS/CTS):** For large packets, the RTS/CTS mechanism can be used to further reduce the likelihood of collisions. The sending device sends an RTS frame to the receiving device, requesting permission to transmit. The receiving device responds with a CTS frame, indicating that it is ready to receive the data. This exchange helps to reserve the channel for the duration of the data transmission.

In the IoT node, based on the ESP32 SoC, MQTT operates within the data link layer to ensure the structured and efficient transmission of sensor data to the gateway. Here's how the implementation typically works:

- **MQTT Client Setup:** The ESP32 is configured as an MQTT client. It connects to an MQTT broker (often hosted on the gateway or a cloud service) and publishes sensor data to specific topics.
- **Data Publishing:** The node collects data from various sensors, formats it into a JSON object, and publishes it to relevant topics on the MQTT broker. Each type of sensor data can be published to a dedicated topic, such as **ph_data** or **n_data**.
- **Quality of Service (QoS):** MQTT supports three levels of Quality of Service (QoS) to ensure reliable message delivery:

- **QoS 0 (At most once):** The message is delivered once with no acknowledgment. This level is suitable for non-critical data where occasional loss is acceptable.
- **QoS 1 (At least once):** The message is delivered at least once, with acknowledgment required from the receiver. This ensures that the message is received but may be duplicated.
- **QoS 2 (Exactly once):** The message is delivered exactly once, with a four-step handshake process ensuring no duplicates. This is the highest level of reliability but involves more overhead.

- **Connection Management:** MQTT uses a keep-alive mechanism to maintain the connection between the node and the broker. This ensures that the node remains connected and can quickly publish new data as it becomes available.
- **Secure Communication:** For enhanced security, MQTT can be configured to use TLS (Transport Layer Security) to encrypt the data transmitted between the node and the gateway, protecting it from eavesdropping and tampering.

Now, MQTT is suitable for the placement of nodes and gateways fall under short to medium range distances, such as WiFi protocol. However, in places where this kind of placement is not suitable, there protocols such as LoRa comes in handy. LoRa (Long Range) technology is a key enabler for long-distance, low-power wireless communication in IoT networks, particularly suited for applications in smart agriculture where sensors and devices are often spread across vast fields. LoRa is a modulation technique derived from chirp spread spectrum (CSS) technology, designed to provide long-range communication at low data rates. Operating in the unlicensed ISM (Industrial, Scientific, and Medical) frequency bands, LoRa offers robust connectivity with minimal power consumption, making it ideal for remote and wide-area sensor networks in agriculture. In a typical smart agriculture setup, IoT nodes equipped with LoRa transceivers collect data from various sensors and transmit this data to a central gateway using LoRa communication. The gateway, in turn, processes the data and forwards it to a cloud server or local system for further analysis. However, a major disadvantage of this paradigm is that it does not have an in-built mechanism in the protocol to follow any multiple access technique. Therefore, if there is collision between messages from two different nodes to a gateway, then there is no method for the gateway to know that there has been a message collision. So, a state machine needs to work on both the node and the gateway to coordinate the sending of messages in case CSMA/CA is used as a MAC protocol, or timing has to be synchronized so that dedicated time slots are alloted to each node to send data.

3.5 | Software Development of the Node

The software development for the IoT node in smart agriculture is carried out using the Arduino IDE, a popular platform for microcontroller programming due to its simplicity and extensive library support. The ESP32 Dev Module is chosen as the Fully Qualified Board Name (FQBN) for this development. This section gives an overview of the software development process, including key initialisations and functionalities, followed by relevant code snippets.

First, the ESP32 Dev Module is selected as the FQBN in the Arduino IDE. The corresponding serial port is chosen to enable communication between the computer and the ESP32 board. The board is connected to the computer system via a USB to UART converter using an FTDI chip, which facilitates programming and serial communication. The RSSAModbusJSONLib.h and RSSAModbusJSONLib.cpp files contain custom functions and initializations for MODBUS communication and data acquisition from sensors. These files define how data is read from the sensors and formatted into JSON objects.

3.5.1 | Initialise: Serial Ports and MQTT

This section describes the initialization of the modules needed to communicate with different peripherals and with the gateway. These include initializing serial ports for UART (for serial monitoring), GNSS (for PNT services), RS485 (for communicating with the sensors), WiFi (to connect with the gateway), LoRa (for long distance transmission), SD Card (for accessing the SD Card) and MQTT (for communication with the gateway).

- **Initialize UART - Serial Monitoring:**
  Here, we first import necessary libraries that include the Arduino Library, Software Serial library and Hardware Serial library. The UART for serial monitoring is initialized in the **void setup()**

function with a baud rate of 9600bps. In this case, the program enables the UART0 port for serial monitoring. Following is a snippet that achieves this:

```
#include <Arduino.h>
#include <SoftwareSerial.h>
#include <HardwareSerial.h>

void setup() {
    Serial.begin(9600);
    Serial.println("Setup␣Complete");
}
```

- **Initialize UART - GNSS:**
The node uses UART2 port for using the GNSS module. Initially, there is a macro that is defined to enable the use of the GNSS module. Then, the General Purpose IO (GPIO) pins 16 and 17 will act as the Receiver (RX) and Transmitter (TX) pins respectively. GPIO 17 is connected to the RX pin of the GNSS module, and GPIO 16 is connected to the TX pin of the GNSS module. First, the serial port for UART2 has to be initialized, followed by initializing the GNSS module and waiting for the fix. Once the fix is there, it will capture the latitude, longitude and the current time in UTC. Following code snippet realizes the same. The code snippet uses the TinyGPS library to communicate with the GNSS module. The GPS module is also used to generate a file name using the day of year in UTC. The functions used are defined as **extractGPSInfoFileName()** and **dayOfYear()**. Some of the variables used in the following snippet are defined in the header file **RSSAModbusJSONLib.h**.

```
#define GPS_ENABLED
// #define GPS_NOT_ENABLED
#include "RSSAModbusJSONLib.h"
#include <Arduino.h>
#include <SoftwareSerial.h>
#include <HardwareSerial.h>
#include <TinyGPS++.h>

TinyGPSPlus gps;
#define GPS_PORT Serial2

int year, month, day, hour, minute, second, doy;

void extractGPSInfoFileName() {
    // Variables to store GPS data

    unsigned long fix_age;

    if (gps.location.isValid() && gps.date.isValid() && gps.time.isValid()) {
    // Extract date and time components
    year = gps.date.year();
    month = gps.date.month();
    day = gps.date.day();
    hour = gps.time.hour();
    minute = gps.time.minute();
    second = gps.time.second();
    }
    // Extract date and time
    // gps.crack_datetime(&year, &month, &day, &hour, &minute, &second, &
fix_age);

    // Convert day, hour, minute, and second to day of year
    doy = dayOfYear(year, month, day);
```

```
    node_doc["year"] = year;
    node_doc["doy"] = doy;
    node_doc["hh"] = hour;
    node_doc["mm"] = minute;
    node_doc["ss"] = second;

    node_doc["lat"] = round(gps.location.lat()*100)/100.0; node_doc["lon"] =
round(gps.location.lng()*100)/100.0;

    // Create a string with the extracted data
    filenameString = "/" + String(site_id) + "_" + String(year) + "_" +
String(doy) + ".json";

    // Print the timestamp string
     Serial.println("filename:␣" + filenameString);
}

// Function to calculate day of year
int dayOfYear(int year, int month, int day) {
    static int days[] = {0,31,59,90,120,151,181,212,243,273,304,334};
    int doy = days[month - 1] + day;
    if (month > 2 && year % 4 == 0 && (year % 100 != 0 || year % 400 == 0))
        doy++;
    return doy;
}

void setup() {
    year = 2024; hour = 0; minute=0; second=0; doy=1;
    Serial.begin(9600);
    GPS_PORT.begin(9600);
    Serial.println("Setup␣Complete");

    #ifdef GPS_ENABLED
    while(1){
        while (GPS_PORT.available() > 0) {
            gps.encode(GPS_PORT.read());
        }
        if (gps.location.isValid()) {
            // Print latitude and longitude
            Serial.print("Latitude:␣");
            Serial.print(gps.location.lat(), 6);
            Serial.print(",␣Longitude:␣");
            Serial.println(gps.location.lng(), 6);

            // Print date and time (if available)
            if (gps.date.isValid() && gps.time.isValid()) {
                Serial.print("Date:␣");
                Serial.print(gps.date.month());
                Serial.print("/");
                Serial.print(gps.date.day());
                Serial.print("/");
                Serial.print(gps.date.year());
                Serial.print("␣Time:␣");
                if (gps.time.hour() < 10) Serial.print('0');
                Serial.print(gps.time.hour());
                Serial.print(":");
                if (gps.time.minute() < 10) Serial.print('0');
```

```
                Serial.print(gps.time.minute());
                Serial.print(":");
                if (gps.time.second() < 10) Serial.print('0');
                Serial.println(gps.time.second());
            }

            // Print number of satellites in view
            Serial.print("Satellites in view: ");
            Serial.println(gps.satellites.value());
            break;
        } else {
            Serial.println("Waiting for valid GPS fix...");
        }
        delay(1000); // Update every second
    }
    #endif }
```

- **Initialize RS485:**
  Here, Data In (DI) and Read Out (RO) of the MAX485 (or a similar interfacing circuit) are connected to GPIO 26 and GPIO 27 of the ESP32 respectively. These are configured as software serial with the GPIO 26 as the transmitter and GPIO 27 as the receiver. These pins work for data transfer between RS485 sensors and the virtual UART port in ESP32. The transmission and reception is controlled with the help of RE and DE pins of the interfacing circuit, where RE is active low and DE is active high. In the case of this node hardware, RE and DE pins are shorted and are connected to the GPIO 25 of the ESP32. The header files **RSSAModbusJSONLib.h** and **RSSAModbusJSONLib.cpp** have the functions that initialize the variables and software serial resources for a sensor. The declaration of the resources are mentioned in the file **RSSAModbusJSONLib.h**, an example of which is shown below for the soil pH sensor:

```
    #ifndef RSSAMODBUSJSONLIB_H
    #define RSSAMODBUSJSONLIB_H

    #include <ArduinoJson.h>
    #include <SoftwareSerial.h>

    #define JSON_DOCUMENT_SIZE 2048  // Adjust the size of each JSON document
    #define TXD_PIN 26
    #define RXD_PIN 27
    #define RE_DE_PIN 25
    #define BUF_SIZE 1024

    extern SoftwareSerial mySerial;

    // Declare JSON objects as extern
    extern StaticJsonDocument<JSON_DOCUMENT_SIZE> node_doc;
    extern StaticJsonDocument<JSON_DOCUMENT_SIZE> gateway_doc_ph;

    // Function declarations
    uint16_t calculateCRC(uint8_t *data, uint8_t length);

    void initialize_node_json(void);

    void initialize_pH(void);
    void acquire_pH(uint8_t slave_id, const char* site_id, const char*
device_id, float lat, float lon, uint8_t year, uint8_t doy, uint8_t hh,
uint8_t mm, uint8_t ss);

    #endif // RSSA_MODBUS_JSON_LIB_H
```

As seen in the above code block, the header file starts out by importing the necessary libraries for JSON object and software serial. Then, the static declarations of the JSON object for datalogging (**node_doc**) and the JSON object for data transmission (**gateway_doc_ph**) are performed. Macros **TXD_PIN, RXD_PIN, RE_DE_PIN** are declared for their connection to DI, RO and the RE and DE pins of the interfacing circuit respectively. This is followed by the declaration to calculate the Cyclic Redundancy Check (CRC) for MODBUS instruction, initialization function for **node_doc** and the data acquisition function for pH.

The following code snippet from **RSSAModbusJSONLib.cpp** have the definitions for the above declared functions:

```
#include "RSSAModbusJSONLib.h"
#include <ArduinoJson.h>
#include <SoftwareSerial.h>

SoftwareSerial mySerial(RXD_PIN, TXD_PIN);

// Define JSON objects as
StaticJsonDocument<JSON_DOCUMENT_SIZE> node_doc;
StaticJsonDocument<JSON_DOCUMENT_SIZE> gateway_doc_ph;


uint16_t calculateCRC(uint8_t *data, uint8_t length)
{
    uint16_t crc = 0xFFFF;
    for (uint8_t i = 0; i < length; i++) {
        crc ^= data[i];
        for (uint8_t j = 0; j < 8; j++) {
            if (crc & 0x0001) {
                crc >>= 1;
                crc ^= 0xA001; // Polynomial used in MODBUS CRC-16
            } else {
                crc >>= 1;
            }
        }
    }
    return crc;
}

void initialize_node_json(void)
{
    node_doc["site_id"] = "RSSA_Site_01";
    node_doc["lat"] = nullptr; node_doc["lon"] = nullptr;
    node_doc["year"] = 2024;
    node_doc["doy"] = nullptr;
    node_doc["hh"] = nullptr;
    node_doc["mm"] = nullptr;
    node_doc["ss"] = nullptr;
    node_doc["ph_sensor"] = "NiuBol_DKI_PH_01"; node_doc["ph_sensor_address"] =
    nullptr;
    node_doc["ph"] = nullptr; node_doc["ph_unit"] = "pH";
    node_doc["npk_sensor"] = "NiuBol_DKI_NPK_01"; node_doc["npk_sensor_address"]
    = nullptr;
    node_doc["npk_n"] = nullptr; node_doc["npk_p"] = nullptr; node_doc["npk_k"] =
     nullptr; node_doc["npk_unit"] = "mg/L";
    node_doc["rain_sensor"] = "NiuBol_DKI_Rain_01"; node_doc["rain_sensor_address"
    ] = nullptr;
```

```
    node_doc["rain"] = nullptr; node_doc["rain_unit"] = "mm";
    node_doc["light_sensor"] = "NiuBol_DKI_Light_01"; node_doc["
    light_sensor_address"] = nullptr;
    node_doc["light"] = nullptr; node_doc["light_unit"] = "Lux";
    node_doc["soil_temp_moisture_ec_sensor"] = "NiuBol_DKI_TEM_01"; node_doc["
    soil_temp_moisture_ec_sensor_address"] = nullptr;
    node_doc["soil_temp"] = nullptr; node_doc["soil_moisture"] = nullptr;
    node_doc["soil_ec"] = nullptr; node_doc["soil_temp_unit"] = "Celsius";
    node_doc["soil_moisture_unit"] = "%VWC"; node_doc["soil_ec_unit"] = "us/cm";
    node_doc["leaf_sensor"] = "NiuBol_DKI_Leaf_01"; node_doc["leaf_sensor_address"
    ] = nullptr;
    node_doc["leaf_temp"] = nullptr; node_doc["leaf_wetness"] = nullptr; node_doc
    ["leaf_temp_unit"] = "Celsius"; node_doc["leaf_wetness_unit"] = "%RH";

}

void initialize_pH(void)
{
    gateway_doc_ph["site_id"] = "RSSA_Site_01";
    gateway_doc_ph["device_id"] = nullptr;
    gateway_doc_ph["lat"] = nullptr; gateway_doc_ph["lon"] = nullptr;
    gateway_doc_ph["year"] = nullptr;
    gateway_doc_ph["doy"] = nullptr;
    gateway_doc_ph["time"] = "00:00:00";
    gateway_doc_ph["Quantity"] = "pH";
    gateway_doc_ph["Value"] = nullptr;
    gateway_doc_ph["Unit"] = "pH";

}
void acquire_pH(uint8_t slave_id, const char* site_id, const char* device_id,
    float lat, float lon, uint8_t year, uint8_t doy, uint8_t hh, uint8_t mm,
    uint8_t ss)
{
    gateway_doc_ph["site_id"] = site_id;
    gateway_doc_ph["device_id"] = device_id;
    gateway_doc_ph["lat"] = lat; gateway_doc_ph["lon"] = lon;
    gateway_doc_ph["year"] = year;
    gateway_doc_ph["doy"] = doy;

    char timeStr[9]; // "hh:mm:ss" + null terminator
    sprintf(timeStr, "%02d:%02d:%02d", hh, mm, ss);
    gateway_doc_ph["time"] = timeStr;

    uint8_t frameData[] = {slave_id, 0x03, 0x00, 0x00, 0x00, 0x01, 0x00, 0x00}; //
     Slave address, Function code, Starting address, Number of registers, CRC Low,
     CRC High
    uint8_t slaveID_RX, opcode_RX, numBytes_RX, data_RX[2];
    uint16_t reading;
    float data_reading;

    node_doc["ph_sensor_address"] = slave_id;

    // Calculate CRC
    uint16_t crc = calculateCRC(frameData, sizeof(frameData) - 2); // Exclude CRC
     bytes from calculation

    // Add CRC to frame data
    frameData[sizeof(frameData) - 2] = crc & 0xFF; // Low byte of CRC
```

```
    frameData[sizeof(frameData) - 1] = (crc >> 8) & 0xFF; // High byte of CRC

    Serial.print("Sending:␣");

    // Enable transmission
    digitalWrite(RE_DE_PIN, HIGH);

    // Send frame over SoftwareSerial
    for (uint8_t i = 0; i < sizeof(frameData); i++) {
        Serial.print("0x"); Serial.print(frameData[i], HEX); Serial.print("␣");
        mySerial.write(frameData[i]);
    }
    Serial.println();

    // Disable transmission
    digitalWrite(RE_DE_PIN, LOW);

    // Print received bytes
    delay(1000); // Wait for response
    uint8_t frameSize_RX = 7, i=0;
    uint8_t incomingByte;
    uint8_t frameRX[BUF_SIZE];
    Serial.print("Receiving:␣");
    while (mySerial.available()) {
        incomingByte = mySerial.read();
        frameRX[i] = incomingByte;
        Serial.print("0x"); Serial.print(frameRX[i], HEX); Serial.print("␣");
        i++;

    }
    char hexString[5];
    if(i>1){
      slaveID_RX = frameRX[0]; opcode_RX = frameRX[1]; numBytes_RX = frameRX[2];
      data_RX[0] = frameRX[3]; data_RX[1] = frameRX[4];
      reading = 0xFFFF & (data_RX[0] << 8); reading = reading | data_RX[1];
      sprintf(hexString, "%04X", reading);
      data_reading = round((float)reading*100)/10000.0;
      node_doc["ph"] = data_reading;
      gateway_doc_ph["Value"] = data_reading;
      Serial.print("Soil␣pH:␣"); Serial.print(data_reading); Serial.print("␣pH\n")
    ;
      Serial.println("␣");
    }
    digitalWrite(RE_DE_PIN, HIGH);

    vTaskDelay(pdMS_TO_TICKS(1000));
}
```

As seen above, **initialize_node_json()** initializes the datalogging fields for the node with their default values as mentioned in section 3.3. Similarly, **initialize_ph()** initializes the JSON object for the data transmission to the gateway. The function **acquire_ph()** frames the instruction to be sent to the sensor, parses the response, and updates the declared JSON objects. It is explained in detail in section 3.5.2. The following code block shows its use in the main code that is uploaded to the ESP32:

```
// Enabling Sensors
#define PH_ENABLED
// #define NPK_ENABLED
```

```
// #define RAIN_ENABLED
// #define LIGHT_ENABLED
// #define SOIL_SEEED_ENABLED
// #define SOIL_NIUBOL_ENABLED
// #define LEAF_ENABLED
#define GPS_ENABLED
// #define GPS_NOT_ENABLED

#include "RSSAModbusJSONLib.h"
#include <Arduino.h>
#include <SoftwareSerial.h>
#include <HardwareSerial.h>
#include <ArduinoJson.h>

#define TXD_PIN 26
#define RXD_PIN 27
#define RE_DE_PIN 25
#define BUF_SIZE 1024

#define MAX_JSON_CHUNK_LEN 100 // Maximum length of each JSON chunk

#define JSON_DOCUMENT_SIZE 2048  // Adjust the size of each JSON document
// Defining Slave IDs
#define PH_SLAVE_ID 1

// Defining Site ID

const char* site_id = "RSSA_BLR_01";

// Defining Device IDs

const char *ph_device_id = "NiuBol_DKI_PH_01";

void setup() {
    year = 2024; hour = 0; minute=0; second=0; doy=1;
    Serial.begin(9600);
    mySerial.begin(9600);
    pinMode(RE_DE_PIN, OUTPUT);
    digitalWrite(RE_DE_PIN, LOW);
    Serial.println("Setup Complete");

  initialize_node_json();

  #ifdef PH_ENABLED
  initialize_pH();
  #endif

}
```

- **Initialize WiFi:**
  To initialize the onboard WiFi functionality of the ESP32, the WiFi library has to be imported into the environment, along with the declaration of the access point and the password of the access point the ESP32 has to be connected to. To do this, in the **setup()** function, a WiFi client has to be declared and initialized, followed by connecting to the WiFi access point.

```
#include <Arduino.h>
#include <WiFi.h>
```

```
// WiFi settings
const char *ssid = "RSSA_AP";      // SSID of the Raspberry Pi AP
const char *password = "RSSA_12345678"; // Password for the Raspberry Pi

WiFiClient espClient;

void setup() {
    year = 2024; hour = 0; minute=0; second=0; doy=1;
    Serial.begin(9600);
    Serial.println("Setup Complete");

    WiFi.begin(ssid, password);
    while (WiFi.status() != WL_CONNECTED) {
        delay(500);
        Serial.println("Connecting to WiFi..");
    }
    Serial.println("Connected to the WiFi network");
    // Print local IP address and start web server
    Serial.println("");
    Serial.println("WiFi connected.");
    Serial.println("IP address: ");
    Serial.println(WiFi.localIP());
}
```

- **Initialize SPI - SD Card:**
  The SD card module is connected to the ESP32 using the SPI protocol, with its Chip Select (**CS**) as the GPIO 5. Initially, a file system is initialized for the SD card, which is then used by the main code in the **loop()** function appending the updated **node_doc** periodically. The following code snippet does the initialization.

```
#include <Arduino.h>
#include <SoftwareSerial.h>
#include <HardwareSerial.h>
#include <SPI.h>
#include <SD.h>
#include <ArduinoJson.h>

#define SD_CS_PIN 5

void setup() {
    Serial.begin(9600);
    Serial.println("Setup Complete");

    if (!SD.begin(SD_CS_PIN)) {
      Serial.println("SD card initialization failed!");
        // return;
    }
  Serial.println("SD card initialized.");
}
```

- **Initialize MQTT:**
  The ESP32 initializes a WiFi client so that it can connect using WiFi to the gateway. Once the connection is established, an MQTT client needs to be setup so that this WiFi connection is used for data transfer. This is done by the following code snippet:

```
#include <Arduino.h>
#include <WiFi.h>
```

```
#include <PubSubClient.h>

// WiFi settings
const char *ssid = "RSSA_AP";     // SSID of the Raspberry Pi AP
const char *password = "RSSA_12345678"; // Password for the Raspberry Pi

// MQTT settings
const char *mqtt_server = "192.168.4.1"; // IP address of the Raspberry Pi
const char *mqtt_topic_pH = "pH_data";
const char *mqtt_topic_n = "n_data";
const char *mqtt_topic_p = "p_data";
const char *mqtt_topic_k = "k_data";
const char *mqtt_topic_rain = "rain_data";
const char *mqtt_topic_light = "light_data";
const char *mqtt_topic_s_temp = "s_temp_data";
const char *mqtt_topic_s_moist = "s_moist_data";
const char *mqtt_topic_s_ec = "s_ec_data";
const char *mqtt_topic_l_wet = "l_wet_data";
const char *mqtt_topic_l_temp = "l_temp_data";
const char* mqtt_username = "rssa-gateway"; // MQTT username
const char* mqtt_password = "rssa_gateway"; // MQTT password
const char* clientID = "RSSA_Node"; // MQTT client ID

WiFiClient espClient;
PubSubClient client(mqtt_server, 1883, espClient);

void reconnect() {
  // Loop until we're reconnected
  int attempts = 0;
  while (!client.connected()) {
    Serial.print("Attempting␣MQTT␣connection...");
    // Attempt to connect
    if (client.connect(clientID, mqtt_username, mqtt_password)) {
      Serial.println("connected");
    } else {
      Serial.print("failed,␣rc=");
      Serial.print(client.state());
      Serial.println("␣try␣again␣in␣5␣seconds");
      attempts++;
      // Wait 5 seconds before retrying
      delay(5000);
    }
    if(attempts >= 5)
      break;
  }
}

void setup() {
    Serial.begin(9600);
    WiFi.begin(ssid, password);
    while (WiFi.status() != WL_CONNECTED) {
        delay(500);
        Serial.println("Connecting␣to␣WiFi..");
    }
    Serial.println("Connected␣to␣the␣WiFi␣network");
    // Print local IP address and start web server
    Serial.println("");
    Serial.println("WiFi␣connected.");
```

```
    Serial.println("IP address: ");
    Serial.println(WiFi.localIP());
    // Set up the MQTT client
    client.setServer(mqtt_server, 1883);
    reconnect();
}
```

3.5.2 | Read Values using RS-485

This section describes the instruction that has to be sent through the software serial to read the data from the sensors. This section also describes how to parse the response and extract the values from the sensors. In the development of the software for this node device, the header files **RSSAModbusJSONLib.h** and **RSSAModbusJSONLib.cpp** have been defined which declare and describe the functionality of reading from sensors using RS-485 sensors, including allocating resources, updating JSON objects, and sending and parsing RS485 instructions.

The node communicates with the sensors using virtual UART interface connected to a circuit such as MAX485 so that the UART interface working on 3.3V TTL can be converted to the RS485 MODBUS protocol. Therefore, the ESP32 has to send instructions in the form of a bitstream, which is converted to RS485 MODBUS signals, and then is transmitted to the sensor. The sensor then responds with RS485 signals, which is converted into a bitstream of 3.3V TTL to be parsed by the ESP32. Typically, the instruction sets for different operations - such as changing slave IDs, writing to the registers inside the sensor, or to read the sensor values - are different depending on the sensor and its manufacturer. However, the instructions follow a fixed format. The reader should note that the instruction to change the Slave ID or to read it can be different. The instructions typically consist of eight 8-bit symbols as shown in Figure 3.4. Their utility is as follows starting from the left in Figure 3.4:

1. **Byte 1 - Slave ID:** This is the unique identifier for the RS485 sensor.
2. **Byte 2 - Opcode:** This is the opcode to do a particular function, such are reading from a register, or writing into a register.
3. **Byte 3 - Starting Register Address (MSB):** This is the Most Significant Byte (MSB) of the starting register address.
4. **Byte 4 - Starting Register Address (LSB):** This is the Least Significant Byte (MSB) of the starting register address.
5. **Byte 5 - Number of registers to read (MSB):** This is the Most Significant Byte (MSB) of the number of registers in the sensor device to be read starting from the start register.
6. **Byte 6 - Number of registers to read (LSB):** This is the Least Significant Byte (LSB) of the number of registers in the sensor device to be read starting from the start register.
7. **Byte 7 - MODBUS CRC Checksum (MSB):** This is the Most Significant Byte (MSB) of the MODBUS CRC Checksum calculated using the function **calculate_crc16(uint8_t *data, uint8_t length)**.
8. **Byte 8 - MODBUS CRC Checksum (LSB):** This is the Least Significant Byte (LSB) of the MODBUS CRC Checksum calculated using the function **calculate_crc16(uint8_t *data, uint8_t length)**.

The following code block defines the function to calculate the CRC-16 MODBUS checksum:

```
uint16_t calculateCRC(uint8_t *data, uint8_t length)
{
    uint16_t crc = 0xFFFF;
    for (uint8_t i = 0; i < length; i++) {
        crc ^= data[i];
        for (uint8_t j = 0; j < 8; j++) {
```

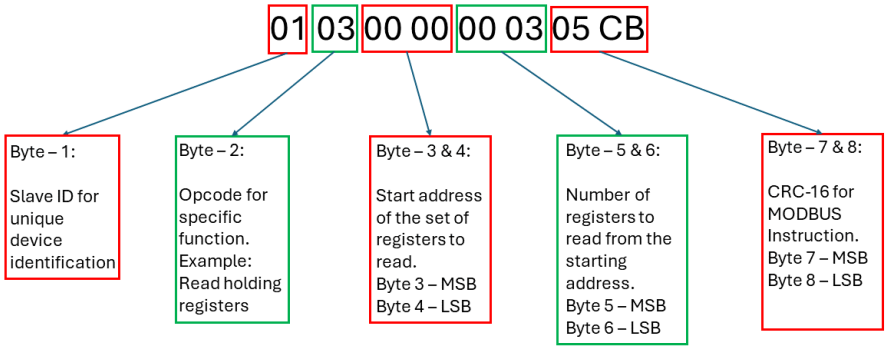

Figure 3.4: MODBUS instruction format

```
            if (crc & 0x0001) {
                crc >>= 1;
                crc ^= 0xA001; // Polynomial used in MODBUS CRC-16
            } else {
                crc >>= 1;
            }
        }
    }
    return crc;
}
```

Figure 3.5 shows a typical response from the sensor when an instruction has been sent to read data acquired by it. As it can be seen, following are the utility of the byte symbols:

1. **Byte 1 - Slave ID:** This is the unique identifier for the RS485 sensor.
2. **Byte 2 - Opcode:** This is the opcode to do a particular function, such are reading from a register, or writing into a register.
3. **Byte 3 - Number of Bytes (N):** This is the total number of bytes that are required to represent the readings from all the registers requested combined. For example, if only one register's value is sought, then the value of N will be 2, since each register stored the value in the form of 16-bit integers, i.e. 2 bytes. Similarly, 2 registers if requested will give a N=4, and so on.
4. **Bytes 4 to 3+N - N/2 16-bit data:** These bytes, when grouped into two bytes, form 16-bit data that are the values of the registers that store the measurements taken by the sensors. These can be converted into true measurements, as they are stored as 16-bit integers, using a conversion factor as described by the manufacturer of the sensor.
5. **Byte 4+N - MODBUS CRC-16 Checksum (MSB):** This is the Most Significant Byte (MSB) of the MODBUS CRC Checksum calculated using the function **calculate_crc16(uint8_t *data, uint8_t length)**.
6. **Byte 5+N - MODBUS CRC-16 Checksum (LSB):** This is the Least Significant Byte (LSB) of the MODBUS CRC Checksum calculated using the function **calculate_crc16(uint8_t *data, uint8_t length)**.

As an example, the following code block sends instruction to the soil pH sensor and parses its response, followed by updating the requisite JSON objects:

```
void acquire_pH(uint8_t slave_id, const char* site_id, const char* device_id, float
    lat, float lon, uint8_t year, uint8_t doy, uint8_t hh, uint8_t mm, uint8_t ss)
{
    gateway_doc_ph["site_id"] = site_id;
    gateway_doc_ph["device_id"] = device_id;
    gateway_doc_ph["lat"] = lat; gateway_doc_ph["lon"] = lon;
```

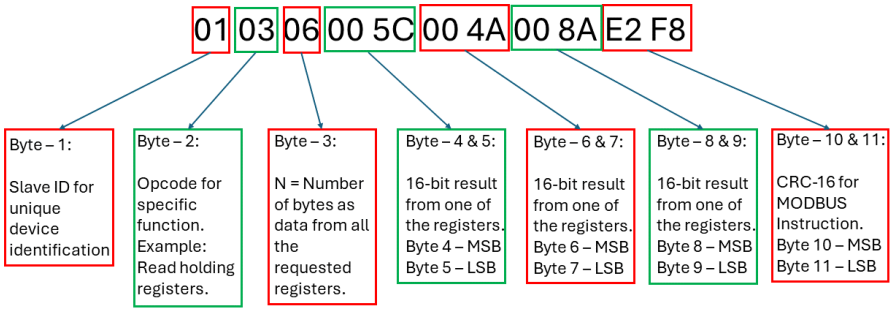

Figure 3.5: Received data in MODBUS format.

```
    gateway_doc_ph["year"] = year;
    gateway_doc_ph["doy"] = doy;

    char timeStr[9]; // "hh:mm:ss" + null terminator
    sprintf(timeStr, "%02d:%02d:%02d", hh, mm, ss);
    gateway_doc_ph["time"] = timeStr;

    uint8_t frameData[] = {slave_id, 0x03, 0x00, 0x00, 0x00, 0x01, 0x00, 0x00}; //
    Slave address, Function code, Starting address, Number of registers, CRC Low, CRC
    High
    uint8_t slaveID_RX, opcode_RX, numBytes_RX, data_RX[2];
    uint16_t reading;
    float data_reading;

    node_doc["ph_sensor_address"] = slave_id;

    // Calculate CRC
    uint16_t crc = calculateCRC(frameData, sizeof(frameData) - 2); // Exclude CRC
    bytes from calculation

    // Add CRC to frame data
    frameData[sizeof(frameData) - 2] = crc & 0xFF; // Low byte of CRC
    frameData[sizeof(frameData) - 1] = (crc >> 8) & 0xFF; // High byte of CRC

    Serial.print("Sending: ");

    // Enable transmission
    digitalWrite(RE_DE_PIN, HIGH);

    // Send frame over SoftwareSerial
    for (uint8_t i = 0; i < sizeof(frameData); i++) {
        Serial.print("0x"); Serial.print(frameData[i], HEX); Serial.print(" ");
        mySerial.write(frameData[i]);
    }
    Serial.println();

    // Disable transmission
    digitalWrite(RE_DE_PIN, LOW);

    // Print received bytes
    delay(1000); // Wait for response
    uint8_t frameSize_RX = 7, i=0;
    uint8_t incomingByte;
    uint8_t frameRX[BUF_SIZE];
```

```
    Serial.print("Receiving:␣");
    while (mySerial.available()) {
        incomingByte = mySerial.read();
        frameRX[i] = incomingByte;
        Serial.print("0x"); Serial.print(frameRX[i], HEX); Serial.print("␣");
        i++;

    }
    char hexString[5];
    if(i>1){
      slaveID_RX = frameRX[0]; opcode_RX = frameRX[1]; numBytes_RX = frameRX[2];
      data_RX[0] = frameRX[3]; data_RX[1] = frameRX[4];
      reading = 0xFFFF & (data_RX[0] << 8); reading = reading | data_RX[1];
      sprintf(hexString, "%04X", reading);
      data_reading = round((float)reading*100)/10000.0;
      node_doc["ph"] = data_reading;
      gateway_doc_ph["Value"] = data_reading;
      Serial.print("Soil␣pH:␣"); Serial.print(data_reading); Serial.print("␣pH\n");
      Serial.println("␣");
    }
    digitalWrite(RE_DE_PIN, HIGH);

    vTaskDelay(pdMS_TO_TICKS(1000));
}
```

3.5.3 | Process Data, Datalogging and Transmit to Gateway

Now, after the data is acquired from the sensors, all the defined JSON objects are updated by the data acquisition functions. The definition and features of all the fields have been discussed in Section 3.3. The JSON objects are now serialised into a JSON string, and then logged locally into a file periodically by the **loop()** function, followed by transmission to the gateway:

```
void loop() {

    #ifdef GPS_ENABLED
    extractGPSInfoFileName();
    #endif

    #ifdef GPS_NOT_ENABLED
    elapsedTime = millis() - startTime;

    // Convert milliseconds to days, hours, minutes, and seconds
    unsigned long seconds = elapsedTime / 1000;
    unsigned long minutes = seconds / 60;
    unsigned long hours = minutes / 60;
    unsigned long doy = hours / 24;

    // Calculate remaining hours, minutes, and seconds
    seconds %= 60;
    minutes %= 60;
    hours %= 24;

    node_doc["doy"] = doy;
    node_doc["hh"] = hours;
    node_doc["mm"] = minutes;
    node_doc["ss"] = seconds;
    #endif
```

```
#ifdef PH_ENABLED
acquire_pH(PH_SLAVE_ID, site_id, ph_device_id, node_doc["lat"], node_doc["lat"],
node_doc["year"], node_doc["doy"], node_doc["hh"], node_doc["mm"], node_doc["ss"]);

#endif

#ifdef NPK_ENABLED
acquire_npk(NPK_SLAVE_ID, site_id, npk_device_id, node_doc["lat"], node_doc["lat"],
 node_doc["year"], node_doc["doy"], node_doc["hh"], node_doc["mm"], node_doc["ss"])
;
#endif

#ifdef RAIN_ENABLED
acquire_rain(RAIN_SLAVE_ID, site_id, rain_device_id, node_doc["lat"], node_doc["
lat"], node_doc["year"], node_doc["doy"], node_doc["hh"], node_doc["mm"], node_doc
["ss"]);
#endif

#ifdef LIGHT_ENABLED
acquire_light(LIGHT_SLAVE_ID, site_id, light_device_id, node_doc["lat"], node_doc["
lat"], node_doc["year"], node_doc["doy"], node_doc["hh"], node_doc["mm"], node_doc
["ss"]);
#endif

#ifdef SOIL_SEEED_ENABLED
acquire_soil_seeed(SOIL_SEEED_SLAVE_ID, site_id, soil_seeed_device_id, node_doc["
lat"], node_doc["lat"], node_doc["year"], node_doc["doy"], node_doc["hh"],
node_doc["mm"], node_doc["ss"]);
#endif

#ifdef SOIL_NIUBOL_ENABLED
acquire_soil_niubol(SOIL_NIUBOL_SLAVE_ID, site_id, soil_niubol_device_id, node_doc
["lat"], node_doc["lat"], node_doc["year"], node_doc["doy"], node_doc["hh"],
node_doc["mm"], node_doc["ss"]);
#endif

#ifdef LEAF_ENABLED
acquire_leaf(LEAF_SLAVE_ID, site_id, leaf_device_id, node_doc["lat"], node_doc["
lat"], node_doc["year"], node_doc["doy"], node_doc["hh"], node_doc["mm"], node_doc
["ss"]);
#endif


// Serialize the JSON object
// Add data to the JSON object

filenameString = +"/" + String(site_id) + "_" + String(year) + "_" + String(doy) +
 ".json";

String jsonString;
char jsonBuffer[2048];
serializeJson(node_doc, jsonString);

// Print the JSON string
Serial.println("JSON␣data:");
Serial.println(jsonString);

Serial.print("File␣Name:␣"); Serial.print(filenameString); Serial.print("\n");
```

```
  // Write the JSON data to a file
  if (!SD.exists(filenameString)) {
  Serial.println("File does not exist. Creating a new file.");
  createNewFile(filenameString.c_str());
}
  appendToFile(filenameString.c_str(), jsonString+"\n");

Serial.println("JSON string appended to file successfully!");

if (!client.connected()) {
  reconnect();
}
client.loop();

#ifdef PH_ENABLED
serializeJson(gateway_doc_ph, jsonBuffer);
client.publish(mqtt_topic_pH, jsonBuffer);
Serial.println("Published: ");
Serial.println(jsonBuffer);
#endif

#ifdef NPK_ENABLED
serializeJson(gateway_doc_n, jsonBuffer);
client.publish(mqtt_topic_n, jsonBuffer);
Serial.println("Published: ");
Serial.println(jsonBuffer);

serializeJson(gateway_doc_p, jsonBuffer);
client.publish(mqtt_topic_p, jsonBuffer);
Serial.println("Published: ");
Serial.println(jsonBuffer);

serializeJson(gateway_doc_k, jsonBuffer);
client.publish(mqtt_topic_k, jsonBuffer);
Serial.println("Published: ");
Serial.println(jsonBuffer);
#endif

#ifdef RAIN_ENABLED
serializeJson(gateway_doc_rain, jsonBuffer);
client.publish(mqtt_topic_rain, jsonBuffer);
Serial.println("Published: ");
Serial.println(jsonBuffer);
#endif

#ifdef LIGHT_ENABLED
serializeJson(gateway_doc_light, jsonBuffer);
client.publish(mqtt_topic_light, jsonBuffer);
Serial.println("Published: ");
Serial.println(jsonBuffer);
#endif

#ifdef SOIL_SEEED_ENABLED
serializeJson(gateway_doc_soil_temp_seeed, jsonBuffer);
client.publish(mqtt_topic_s_temp, jsonBuffer);
Serial.println("Published: ");
Serial.println(jsonBuffer);
```

```
  serializeJson(gateway_doc_soil_moist_seeed, jsonBuffer);
  client.publish(mqtt_topic_s_moist, jsonBuffer);
  Serial.println("Published:␣");
  Serial.println(jsonBuffer);

  serializeJson(gateway_doc_soil_ec_seeed, jsonBuffer);
  client.publish(mqtt_topic_s_ec, jsonBuffer);
  Serial.println("Published:␣");
  Serial.println(jsonBuffer);
  #endif

  #ifdef SOIL_NIUBOL_ENABLED
  serializeJson(gateway_doc_soil_temp_niubol, jsonBuffer);
  client.publish(mqtt_topic_s_temp, jsonBuffer);
  Serial.println("Published:␣");
  Serial.println(jsonBuffer);

  serializeJson(gateway_doc_soil_moist_niubol, jsonBuffer);
  client.publish(mqtt_topic_s_moist, jsonBuffer);
  Serial.println("Published:␣");
  Serial.println(jsonBuffer);

  serializeJson(gateway_doc_soil_ec_niubol, jsonBuffer);
  client.publish(mqtt_topic_s_ec, jsonBuffer);
  Serial.println("Published:␣");
  Serial.println(jsonBuffer);
  #endif

  #ifdef LEAF_ENABLED
  serializeJson(gateway_doc_leaf_temp, jsonBuffer);
  client.publish(mqtt_topic_l_temp, jsonBuffer);
  Serial.println("Published:␣");
  Serial.println(jsonBuffer);

  serializeJson(gateway_doc_leaf_wet, jsonBuffer);
  client.publish(mqtt_topic_l_wet, jsonBuffer);
  Serial.println("Published:␣");
  Serial.println(jsonBuffer);
  #endif
}
```

As seen in the above code block, the function **extractGPSInfoFileName()** is called when the GPS is enabled. This uses the GPS day of year and time and creates the file name and timestamps the JSON objects using GPS as seen below:

```
void extractGPSInfoFileName() {
  // Variables to store GPS data

  unsigned long fix_age;

    if (gps.location.isValid() && gps.date.isValid() && gps.time.isValid()) {
    // Extract date and time components
    year = gps.date.year();
    month = gps.date.month();
    day = gps.date.day();
    hour = gps.time.hour();
    minute = gps.time.minute();
    second = gps.time.second();
```

```
    }
  // Extract date and time
  // gps.crack_datetime(&year, &month, &day, &hour, &minute, &second, &fix_age);

  // Convert day, hour, minute, and second to day of year
  doy = dayOfYear(year, month, day);

  node_doc["year"] = year;
  node_doc["doy"] = doy;
  node_doc["hh"] = hour;
  node_doc["mm"] = minute;
  node_doc["ss"] = second;

  node_doc["lat"] = round(gps.location.lat()*100)/100.0; node_doc["lon"] = round(gps.
    location.lng()*100)/100.0;

  // Create a string with the extracted data
  filenameString = "/" + String(site_id) + "_" + String(year) + "_" + String(doy) + ".
    json";

  // Print the timestamp string
  Serial.println("filename:␣" + filenameString);
}

// Function to calculate day of year
int dayOfYear(int year, int month, int day) {
  static int days[] = {0,31,59,90,120,151,181,212,243,273,304,334};
  int doy = days[month - 1] + day;
  if (month > 2 && year % 4 == 0 && (year % 100 != 0 || year % 400 == 0))
    doy++;
  return doy;
}
```

Now, the ESP32 checks if a file by the extracted file name exists or not in the SD card file system. If it exists, it appends the JSON object to the file by calling the **appendToFile(const char* filename, String data)** function. If the file does not exist, then the **createNewFile(const char* filename)** function is called, followed by the **appendToFile(const char* filename, String data)** function call. Their definition are as follows:

```
void createNewFile(const char* filename) {
  // Open the file in write mode
  File file = SD.open(filename, FILE_WRITE);

  // If the file opened successfully
  if (file) {
    // Close the file
    file.close();
    Serial.println("File␣created␣successfully.");
  } else {
    Serial.println("Error␣creating␣file.");
  }
}

void appendToFile(const char* filename, String data) {
  // Open the file in append mode
  File file = SD.open(filename, FILE_APPEND);

  // If the file opened successfully
  if (file) {
```

```
    // Append data to the file
    file.print(data);

    // Close the file
    file.close();
  } else {
    Serial.println("Error␣appending␣to␣file.");
  }
}
```

After storing the data locally, the data is transmitted to the gateway under different topics defined in the node. In case the connection has terminated before the start of transmission, the **reconnect()** function is called which persistently attempts to connect to the MQTT broker till a fixed number of attempts.

```
void reconnect() {
  // Loop until we're reconnected
  int attempts = 0;
  while (!client.connected()) {
    Serial.print("Attempting␣MQTT␣connection...");
    // Attempt to connect
    if (client.connect(clientID, mqtt_username, mqtt_password)) {
      Serial.println("connected");
    } else {
      Serial.print("failed,␣rc=");
      Serial.print(client.state());
      Serial.println("␣try␣again␣in␣5␣seconds");
      attempts++;
      // Wait 5 seconds before retrying
      delay(5000);
    }
    if(attempts >= 5)
      break;
  }
}
```

3.5.4 | Final Code

Following code block shows the final look of the RSSAModbusJSONLib.h header file:

```
#ifndef RSSAMODBUSJSONLIB_H
#define RSSAMODBUSJSONLIB_H

#include <ArduinoJson.h>
#include <SoftwareSerial.h>

#define JSON_DOCUMENT_SIZE 2048  // Adjust the size of each JSON document
#define TXD_PIN 26     // RS485 TX
#define RXD_PIN 27     // RS485 RX
#define RE_DE_PIN 25    //RS485 RE DE
#define BUF_SIZE 1024

extern SoftwareSerial mySerial;    //Software Serial for RS485

// Declare JSON objects as extern
extern StaticJsonDocument<JSON_DOCUMENT_SIZE> node_doc;    //JSON object for local
    datalogging
extern StaticJsonDocument<JSON_DOCUMENT_SIZE> gateway_doc_ph;    //JSON object for
    transmitting Soil pH
extern StaticJsonDocument<JSON_DOCUMENT_SIZE> gateway_doc_n;    //JSON object for
    transmitting Soil Nitrogen
```

```
extern StaticJsonDocument<JSON_DOCUMENT_SIZE> gateway_doc_p;    //JSON object for
    transmitting Soil Phosphorous
extern StaticJsonDocument<JSON_DOCUMENT_SIZE> gateway_doc_k;    //JSON object for
    transmitting Soil Potassium
extern StaticJsonDocument<JSON_DOCUMENT_SIZE> gateway_doc_rain;    //JSON object for
    transmitting Rain
extern StaticJsonDocument<JSON_DOCUMENT_SIZE> gateway_doc_light;    //JSON object for
    transmitting Light
extern StaticJsonDocument<JSON_DOCUMENT_SIZE> gateway_doc_soil_temp_seeed;    //JSON
    object for transmitting Soil Temp
extern StaticJsonDocument<JSON_DOCUMENT_SIZE> gateway_doc_soil_moist_seeed;    //JSON
    object for transmitting Soil Moisture
extern StaticJsonDocument<JSON_DOCUMENT_SIZE> gateway_doc_soil_ec_seeed;    //JSON
    object for transmitting Soil EC
extern StaticJsonDocument<JSON_DOCUMENT_SIZE> gateway_doc_soil_temp_niubol;    //JSON
    object for transmitting Soil Temp
extern StaticJsonDocument<JSON_DOCUMENT_SIZE> gateway_doc_soil_moist_niubol;    //JSON
     object for transmitting Soil Moisture
extern StaticJsonDocument<JSON_DOCUMENT_SIZE> gateway_doc_soil_ec_niubol;    //JSON
    object for transmitting Soil EC
extern StaticJsonDocument<JSON_DOCUMENT_SIZE> gateway_doc_leaf_temp;    //JSON object
    for transmitting Leaf Temp
extern StaticJsonDocument<JSON_DOCUMENT_SIZE> gateway_doc_leaf_wet;    //JSON object
    for transmitting Leaf Wetness
// Function declarations

/*
 * Function: calculateCRC
 * ------------------
 * Calculate CRC for MODBUS Instruction
 *
 * Parameters:
 *   data (uint8_t [])  - The  MODBUS instruction frame
 *   length (uint8_t) - Number of bytes in the instruction frame
 *
 * Returns:
 *   uint16_t
 *
 * Author:
 *   Swastik Bhattacharya
 *
 * Date:
 *   July 30, 2024
 */
uint16_t calculateCRC(uint8_t *data, uint8_t length);

/*
 * Function: initialize_node_json
 * ------------------
 * Initialize JSON object for local datalogging
 *
 * Parameters:
 *   void
 *
 * Returns:
 *   void
 *
 * Author:
```

```
 *   Swastik Bhattacharya
 *
 * Date:
 *   July 30, 2024
 */
void initialize_node_json(void);

/*
 * Function: initialize_pH
 * -----------------
 * Initialize JSON object to transmit Soil pH
 *
 * Parameters:
 *   void
 *
 * Returns:
 *   void
 *
 * Author:
 *   Swastik Bhattacharya
 *
 * Date:
 *   July 30, 2024
 */
void initialize_pH(void);

/*
 * Function: acquire_pH
 * -----------------
 * Initialize JSON object to acquire pH values from the sensor and store in JSONs
 *
 * Parameters:
 *   slave_id (uint8_t) - Unique Slave ID for the RS485 sensor
 *   site_id (const char*) - ID of the site the node is installed
 *   device_id (const char*) - ID for the sensor taking data
 *   lat (float) - Latitude
 *   lon (float) - Longitude
 *   year (uint8_t) - Year of the measurement taken
 *   doy (uint8_t) - Day of year when the measurement is taken
 *   hh (uint8_t) - Timestamp hour
 *   mm (uint8_t) - Timestamp minutes
 *   ss (uint8_t) - Timestamp seconds
 *
 * Returns:
 *   void
 *
 * Author:
 *   Swastik Bhattacharya
 *
 * Date:
 *   July 30, 2024
 */
void acquire_pH(uint8_t slave_id, const char* site_id, const char* device_id, float
    lat, float lon, uint8_t year, uint8_t doy, uint8_t hh, uint8_t mm, uint8_t ss);

/*
 * Function: initialize_npk
 * -----------------
```

```
 * Initialize JSON object to transmit Soil NPK
 *
 * Parameters:
 *   void
 *
 * Returns:
 *   void
 *
 * Author:
 *   Swastik Bhattacharya
 *
 * Date:
 *   July 30, 2024
 */
void initialize_npk(void);

/*
 * Function: acquire_npk
 * ---------------------
 * Initialize JSON object to acquire NPK values from the sensor and store in JSONs
 *
 * Parameters:
 *   slave_id (uint8_t) - Unique Slave ID for the RS485 sensor
 *   site_id (const char*) - ID of the site the node is installed
 *   device_id (const char*) - ID for the sensor taking data
 *   lat (float) - Latitude
 *   lon (float) - Longitude
 *   year (uint8_t) - Year of the measurement taken
 *   doy (uint8_t) - Day of year when the measurement is taken
 *   hh (uint8_t) - Timestamp hour
 *   mm (uint8_t) - Timestamp minutes
 *   ss (uint8_t) - Timestamp seconds
 *
 * Returns:
 *   void
 *
 * Author:
 *   Swastik Bhattacharya
 *
 * Date:
 *   July 30, 2024
 */
void acquire_npk(uint8_t slave_id, const char* site_id, const char* device_id, float
    lat, float lon, uint8_t year, uint8_t doy, uint8_t hh, uint8_t mm, uint8_t ss);

/*
 * Function: initialize_rain
 * ---------------------
 * Initialize JSON object to transmit Rain
 *
 * Parameters:
 *   void
 *
 * Returns:
 *   void
 *
 * Author:
 *   Swastik Bhattacharya
```

```
 *
 * Date:
 *   July 30, 2024
 */
void initialize_rain(void);

/*
 * Function: acquire_rain
 * ------------------
 * Initialize JSON object to acquire Rain values from the sensor and store in JSONs
 *
 * Parameters:
 *   slave_id (uint8_t) - Unique Slave ID for the RS485 sensor
 *   site_id (const char*) - ID of the site the node is installed
 *   device_id (const char*) - ID for the sensor taking data
 *   lat (float) - Latitude
 *   lon (float) - Longitude
 *   year (uint8_t) - Year of the measurement taken
 *   doy (uint8_t) - Day of year when the measurement is taken
 *   hh (uint8_t) - Timestamp hour
 *   mm (uint8_t) - Timestamp minutes
 *   ss (uint8_t) - Timestamp seconds
 *
 * Returns:
 *   void
 *
 * Author:
 *   Swastik Bhattacharya
 *
 * Date:
 *   July 30, 2024
 */
void acquire_rain(uint8_t slave_id, const char* site_id, const char* device_id, float
    lat, float lon, uint8_t year, uint8_t doy, uint8_t hh, uint8_t mm, uint8_t ss);

/*
 * Function: initialize_light
 * ------------------
 * Initialize JSON object to transmit Light
 *
 * Parameters:
 *   void
 *
 * Returns:
 *   void
 *
 * Author:
 *   Swastik Bhattacharya
 *
 * Date:
 *   July 30, 2024
 */
void initialize_light(void);

/*
 * Function: acquire_light
 * ------------------
 * Initialize JSON object to acquire Light values from the sensor and store in JSONs
```

```
 *
 * Parameters:
 *   slave_id (uint8_t) - Unique Slave ID for the RS485 sensor
 *   site_id (const char*) - ID of the site the node is installed
 *   device_id (const char*) - ID for the sensor taking data
 *   lat (float) - Latitude
 *   lon (float) - Longitude
 *   year (uint8_t) - Year of the measurement taken
 *   doy (uint8_t) - Day of year when the measurement is taken
 *   hh (uint8_t) - Timestamp hour
 *   mm (uint8_t) - Timestamp minutes
 *   ss (uint8_t) - Timestamp seconds
 *
 * Returns:
 *   void
 *
 * Author:
 *   Swastik Bhattacharya
 *
 * Date:
 *   July 30, 2024
 */
void acquire_light(uint8_t slave_id, const char* site_id, const char* device_id, float
    lat, float lon, uint8_t year, uint8_t doy, uint8_t hh, uint8_t mm, uint8_t ss);

/*
 * Function: initialize_soil_seeed
 * -------------------
 * Initialize JSON object to transmit Soil TEM using Seeed Studio sensor
 *
 * Parameters:
 *   void
 *
 * Returns:
 *   void
 *
 * Author:
 *   Swastik Bhattacharya
 *
 * Date:
 *   July 30, 2024
 */
void initialize_soil_seeed(void);

/*
 * Function: acquire_soil_seeed
 * -------------------
 * Initialize JSON object to acquire Soil TEM values using Seeed Studio Sensor from
    the sensor and store in JSONs
 *
 * Parameters:
 *   slave_id (uint8_t) - Unique Slave ID for the RS485 sensor
 *   site_id (const char*) - ID of the site the node is installed
 *   device_id (const char*) - ID for the sensor taking data
 *   lat (float) - Latitude
 *   lon (float) - Longitude
 *   year (uint8_t) - Year of the measurement taken
 *   doy (uint8_t) - Day of year when the measurement is taken
```

```
 *   hh (uint8_t) - Timestamp hour
 *   mm (uint8_t) - Timestamp minutes
 *   ss (uint8_t) - Timestamp seconds
 *
 * Returns:
 *   void
 *
 * Author:
 *   Swastik Bhattacharya
 *
 * Date:
 *   July 30, 2024
 */
void acquire_soil_seeed(uint8_t slave_id, const char* site_id, const char* device_id,
    float lat, float lon, uint8_t year, uint8_t doy, uint8_t hh, uint8_t mm, uint8_t
    ss);

/*
 * Function: initialize_soil_niubol
 * ---------------------
 * Initialize JSON object to transmit Soil TEM using NiuBol sensor
 *
 * Parameters:
 *   void
 *
 * Returns:
 *   void
 *
 * Author:
 *   Swastik Bhattacharya
 *
 * Date:
 *   July 30, 2024
 */
void initialize_soil_niubol(void);

/*
 * Function: acquire_soil_niubol
 * ---------------------
 * Initialize JSON object to acquire Soil TEM values using NiuBol Sensor from the
    sensor and store in JSONs
 *
 * Parameters:
 *   slave_id (uint8_t) - Unique Slave ID for the RS485 sensor
 *   site_id (const char*) - ID of the site the node is installed
 *   device_id (const char*) - ID for the sensor taking data
 *   lat (float) - Latitude
 *   lon (float) - Longitude
 *   year (uint8_t) - Year of the measurement taken
 *   doy (uint8_t) - Day of year when the measurement is taken
 *   hh (uint8_t) - Timestamp hour
 *   mm (uint8_t) - Timestamp minutes
 *   ss (uint8_t) - Timestamp seconds
 *
 * Returns:
 *   void
 *
 * Author:
```

```
 *   Swastik Bhattacharya
 *
 * Date:
 *   July 30, 2024
 */
void acquire_soil_niubol(uint8_t slave_id, const char* site_id, const char* device_id,
    float lat, float lon, uint8_t year, uint8_t doy, uint8_t hh, uint8_t mm, uint8_t
   ss);

/*
 * Function: initialize_leaf
 * -------------------
 * Initialize JSON object to transmit Leaf Temp and Wetness
 *
 * Parameters:
 *   void
 *
 * Returns:
 *   void
 *
 * Author:
 *   Swastik Bhattacharya
 *
 * Date:
 *   July 30, 2024
 */
void initialize_leaf(void);

/*
 * Function: acquire_soil_niubol
 * -------------------
 * Initialize JSON object to acquire Leaf Temperature and Wetness from the sensor and
   store in JSONs
 *
 * Parameters:
 *   slave_id (uint8_t) - Unique Slave ID for the RS485 sensor
 *   site_id (const char*) - ID of the site the node is installed
 *   device_id (const char*) - ID for the sensor taking data
 *   lat (float) - Latitude
 *   lon (float) - Longitude
 *   year (uint8_t) - Year of the measurement taken
 *   doy (uint8_t) - Day of year when the measurement is taken
 *   hh (uint8_t) - Timestamp hour
 *   mm (uint8_t) - Timestamp minutes
 *   ss (uint8_t) - Timestamp seconds
 *
 * Returns:
 *   void
 *
 * Author:
 *   Swastik Bhattacharya
 *
 * Date:
 *   July 30, 2024
 */
void acquire_leaf(uint8_t slave_id, const char* site_id, const char* device_id, float
   lat, float lon, uint8_t year, uint8_t doy, uint8_t hh, uint8_t mm, uint8_t ss);
```

```
#endif // RSSA_MODBUS_JSON_LIB_H
```

The following code block shows the final look of the RSSAModbusJSONLib.cpp file with all the function definitions:

```
#include "RSSAModbusJSONLib.h"
#include <ArduinoJson.h>
#include <SoftwareSerial.h>

SoftwareSerial mySerial(RXD_PIN, TXD_PIN);

// Define JSON objects as
StaticJsonDocument<JSON_DOCUMENT_SIZE> node_doc;
StaticJsonDocument<JSON_DOCUMENT_SIZE> gateway_doc_ph;
StaticJsonDocument<JSON_DOCUMENT_SIZE> gateway_doc_n;
StaticJsonDocument<JSON_DOCUMENT_SIZE> gateway_doc_p;
StaticJsonDocument<JSON_DOCUMENT_SIZE> gateway_doc_k;
StaticJsonDocument<JSON_DOCUMENT_SIZE> gateway_doc_rain;
StaticJsonDocument<JSON_DOCUMENT_SIZE> gateway_doc_light;
StaticJsonDocument<JSON_DOCUMENT_SIZE> gateway_doc_soil_temp_seeed;
StaticJsonDocument<JSON_DOCUMENT_SIZE> gateway_doc_soil_moist_seeed;
StaticJsonDocument<JSON_DOCUMENT_SIZE> gateway_doc_soil_ec_seeed;
StaticJsonDocument<JSON_DOCUMENT_SIZE> gateway_doc_soil_temp_niubol;
StaticJsonDocument<JSON_DOCUMENT_SIZE> gateway_doc_soil_moist_niubol;
StaticJsonDocument<JSON_DOCUMENT_SIZE> gateway_doc_soil_ec_niubol;
StaticJsonDocument<JSON_DOCUMENT_SIZE> gateway_doc_leaf_temp;
StaticJsonDocument<JSON_DOCUMENT_SIZE> gateway_doc_leaf_wet;


uint16_t calculateCRC(uint8_t *data, uint8_t length)
{
    uint16_t crc = 0xFFFF;
    for (uint8_t i = 0; i < length; i++) {
        crc ^= data[i];
        for (uint8_t j = 0; j < 8; j++) {
            if (crc & 0x0001) {
                crc >>= 1;
                crc ^= 0xA001; // Polynomial used in MODBUS CRC-16
            } else {
                crc >>= 1;
            }
        }
    }
    return crc;
}

void initialize_node_json(void)
{
    node_doc["site_id"] = "RSSA_Site_01";
    node_doc["lat"] = nullptr; node_doc["lon"] = nullptr;
    node_doc["year"] = 2024;
    node_doc["doy"] = nullptr;
    node_doc["hh"] = nullptr;
    node_doc["mm"] = nullptr;
    node_doc["ss"] = nullptr;
    node_doc["ph_sensor"] = "NiuBol_DKI_PH_01"; node_doc["ph_sensor_address"] =
    nullptr;
    node_doc["ph"] = nullptr; node_doc["ph_unit"] = "pH";
    node_doc["npk_sensor"] = "NiuBol_DKI_NPK_01"; node_doc["npk_sensor_address"] =
```

```
    nullptr;
    node_doc["npk_n"] = nullptr; node_doc["npk_p"] = nullptr; node_doc["npk_k"] =
    nullptr; node_doc["npk_unit"] = "mg/L";
    node_doc["rain_sensor"] = "NiuBol_DKI_Rain_01"; node_doc["rain_sensor_address"] =
    nullptr;
    node_doc["rain"] = nullptr; node_doc["rain_unit"] = "mm";
    node_doc["light_sensor"] = "NiuBol_DKI_Light_01"; node_doc["light_sensor_address"]
     = nullptr;
    node_doc["light"] = nullptr; node_doc["light_unit"] = "Lux";
    node_doc["soil_temp_moisture_ec_sensor"] = "NiuBol_DKI_TEM_01"; node_doc["
    soil_temp_moisture_ec_sensor_address"] = nullptr;
    node_doc["soil_temp"] = nullptr; node_doc["soil_moisture"] = nullptr; node_doc["
    soil_ec"] = nullptr; node_doc["soil_temp_unit"] = "Celsius"; node_doc["
    soil_moisture_unit"] = "%VWC"; node_doc["soil_ec_unit"] = "us/cm";
    node_doc["leaf_sensor"] = "NiuBol_DKI_Leaf_01"; node_doc["leaf_sensor_address"] =
    nullptr;
    node_doc["leaf_temp"] = nullptr; node_doc["leaf_wetness"] = nullptr; node_doc["
    leaf_temp_unit"] = "Celsius"; node_doc["leaf_wetness_unit"] = "%RH";

}

void initialize_pH(void)
{
    gateway_doc_ph["site_id"] = "RSSA_Site_01";
    gateway_doc_ph["device_id"] = nullptr;
    gateway_doc_ph["lat"] = nullptr; gateway_doc_ph["lon"] = nullptr;
    gateway_doc_ph["year"] = nullptr;
    gateway_doc_ph["doy"] = nullptr;
    gateway_doc_ph["time"] = "00:00:00";
    gateway_doc_ph["Quantity"] = "pH";
    gateway_doc_ph["Value"] = nullptr;
    gateway_doc_ph["Unit"] = "pH";

}
void acquire_pH(uint8_t slave_id, const char* site_id, const char* device_id, float
    lat, float lon, uint8_t year, uint8_t doy, uint8_t hh, uint8_t mm, uint8_t ss)
{
    gateway_doc_ph["site_id"] = site_id;
    gateway_doc_ph["device_id"] = device_id;
    gateway_doc_ph["lat"] = lat; gateway_doc_ph["lon"] = lon;
    gateway_doc_ph["year"] = year;
    gateway_doc_ph["doy"] = doy;

    char timeStr[9]; // "hh:mm:ss" + null terminator
    sprintf(timeStr, "%02d:%02d:%02d", hh, mm, ss);
    gateway_doc_ph["time"] = timeStr;

    uint8_t frameData[] = {slave_id, 0x03, 0x00, 0x00, 0x00, 0x01, 0x00, 0x00}; //
    Slave address, Function code, Starting address, Number of registers, CRC Low, CRC
    High
    uint8_t slaveID_RX, opcode_RX, numBytes_RX, data_RX[2];
    uint16_t reading;
    float data_reading;

    node_doc["ph_sensor_address"] = slave_id;

    // Calculate CRC
    uint16_t crc = calculateCRC(frameData, sizeof(frameData) - 2); // Exclude CRC
```

```
    bytes from calculation

    // Add CRC to frame data
    frameData[sizeof(frameData) - 2] = crc & 0xFF; // Low byte of CRC
    frameData[sizeof(frameData) - 1] = (crc >> 8) & 0xFF; // High byte of CRC

    Serial.print("Sending:␣");

    // Enable transmission
    digitalWrite(RE_DE_PIN, HIGH);

    // Send frame over SoftwareSerial
    for (uint8_t i = 0; i < sizeof(frameData); i++) {
        Serial.print("0x"); Serial.print(frameData[i], HEX); Serial.print("␣");
        mySerial.write(frameData[i]);
    }
    Serial.println();

    // Disable transmission
    digitalWrite(RE_DE_PIN, LOW);

    // Print received bytes
    delay(1000); // Wait for response
    uint8_t frameSize_RX = 7, i=0;
    uint8_t incomingByte;
    uint8_t frameRX[BUF_SIZE];
    Serial.print("Receiving:␣");
    while (mySerial.available()) {
        incomingByte = mySerial.read();
        frameRX[i] = incomingByte;
        Serial.print("0x"); Serial.print(frameRX[i], HEX); Serial.print("␣");
        i++;

    }
    char hexString[5];
    if(i>1){
      slaveID_RX = frameRX[0]; opcode_RX = frameRX[1]; numBytes_RX = frameRX[2];
      data_RX[0] = frameRX[3]; data_RX[1] = frameRX[4];
      reading = 0xFFFF & (data_RX[0] << 8); reading = reading | data_RX[1];
      sprintf(hexString, "%04X", reading);
      data_reading = round((float)reading*100)/10000.0;
      node_doc["ph"] = data_reading;
      gateway_doc_ph["Value"] = data_reading;
      Serial.print("Soil␣pH:␣"); Serial.print(data_reading); Serial.print("␣pH\n");
      Serial.println("␣");
    }
    digitalWrite(RE_DE_PIN, HIGH);

    vTaskDelay(pdMS_TO_TICKS(1000));
}

void initialize_npk(void)
{
    gateway_doc_n["site_id"] = "RSSA_Site_01";
    gateway_doc_n["device_id"] = nullptr;
    gateway_doc_n["lat"] = nullptr; gateway_doc_n["lon"] = nullptr;
    gateway_doc_n["year"] = nullptr;
    gateway_doc_n["doy"] = nullptr;
```

```
    gateway_doc_n["time"] = "00:00:00";
    gateway_doc_n["Quantity"] = "Nitrogen";
    gateway_doc_n["Value"] = nullptr;
    gateway_doc_n["Unit"] = "mg/L";

    gateway_doc_p["site_id"] = "RSSA_Site_01";
    gateway_doc_p["device_id"] = nullptr;
    gateway_doc_p["lat"] = nullptr; gateway_doc_p["lon"] = nullptr;
    gateway_doc_p["year"] = nullptr;
    gateway_doc_p["doy"] = nullptr;
    gateway_doc_p["time"] = "00:00:00";
    gateway_doc_p["Quantity"] = "Phosphorous";
    gateway_doc_p["Value"] = nullptr;
    gateway_doc_p["Unit"] = "mg/L";

    gateway_doc_k["site_id"] = "RSSA_Site_01";
    gateway_doc_k["device_id"] = nullptr;
    gateway_doc_k["lat"] = nullptr; gateway_doc_k["lon"] = nullptr;
    gateway_doc_k["year"] = nullptr;
    gateway_doc_k["doy"] = nullptr;
    gateway_doc_k["time"] = "00:00:00";
    gateway_doc_k["Quantity"] = "Potassium";
    gateway_doc_k["Value"] = nullptr;
    gateway_doc_k["Unit"] = "mg/L";
}
void acquire_npk(uint8_t slave_id, const char* site_id, const char* device_id, float
    lat, float lon, uint8_t year, uint8_t doy, uint8_t hh, uint8_t mm, uint8_t ss)
{
    gateway_doc_n["site_id"] = site_id;  gateway_doc_p["site_id"] = site_id;
    gateway_doc_k["site_id"] = site_id;
    gateway_doc_n["device_id"] = device_id; gateway_doc_p["device_id"] = device_id;
    gateway_doc_k["device_id"] = device_id;
    gateway_doc_n["lat"] = lat; gateway_doc_n["lon"] = lon; gateway_doc_p["lat"] = lat;
     gateway_doc_p["lon"] = lon; gateway_doc_k["lat"] = lat; gateway_doc_k["lon"] =
    lon;
    gateway_doc_n["year"] = year; gateway_doc_p["year"] = year; gateway_doc_k["year"]
    = year;
    gateway_doc_n["doy"] = doy; gateway_doc_p["doy"] = doy; gateway_doc_k["doy"] = doy;


    char timeStr[9]; // "hh:mm:ss" + null terminator
    sprintf(timeStr, "%02d:%02d:%02d", hh, mm, ss);
    gateway_doc_n["time"] = timeStr; gateway_doc_p["time"] = timeStr; gateway_doc_k["
    time"] = timeStr;

    uint8_t frameData[] = {slave_id, 0x03, 0x00, 0x00, 0x00, 0x03, 0x00, 0x00}; //
    Slave address, Function code, Starting address, Number of registers, CRC Low, CRC
    High
    uint8_t slaveID_RX, opcode_RX, numBytes_RX, data_RX[2];
    uint16_t reading;
    float data_reading;

    node_doc["npk_sensor_address"] = slave_id;

    // Calculate CRC
    uint16_t crc = calculateCRC(frameData, sizeof(frameData) - 2); // Exclude CRC
    bytes from calculation
```

```
// Add CRC to frame data
frameData[sizeof(frameData) - 2] = crc & 0xFF; // Low byte of CRC
frameData[sizeof(frameData) - 1] = (crc >> 8) & 0xFF; // High byte of CRC

Serial.print("Sending: ");

// Enable transmission
digitalWrite(RE_DE_PIN, HIGH);

// Send frame over SoftwareSerial
for (uint8_t i = 0; i < sizeof(frameData); i++) {
  Serial.print("0x"); Serial.print(frameData[i], HEX); Serial.print(" ");
    mySerial.write(frameData[i]);
}

// Disable transmission
digitalWrite(RE_DE_PIN, LOW);

// Print received bytes
vTaskDelay(pdMS_TO_TICKS(1000)); // Wait for response
uint8_t frameSize_RX = 11, i=0;
uint8_t frameRX[BUF_SIZE];
uint8_t incomingByte;
Serial.print("Receiving: ");
while (mySerial.available()) {
    incomingByte = mySerial.read();
    frameRX[i] = incomingByte;
    Serial.print("0x"); Serial.print(frameRX[i], HEX); Serial.print(" ");
    i++;

}
char hexString[5];
if(i>1){
  slaveID_RX = frameRX[0]; opcode_RX = frameRX[1]; numBytes_RX = frameRX[2];

  data_RX[0] = frameRX[3]; data_RX[1] = frameRX[4];
  reading = 0xFFFF & (data_RX[0] << 8); reading = reading | data_RX[1];
  sprintf(hexString, "%04X", reading);
  data_reading = round((float)reading*100)/10000.0;
  gateway_doc_n["Value"] = data_reading;
  node_doc["npk_n"] = data_reading;
  Serial.print("Nitrogen: ");
  Serial.print(data_reading);
  Serial.print(" mg/L\n");

  data_RX[0] = frameRX[5]; data_RX[1] = frameRX[6];
  reading = 0xFFFF & (data_RX[0] << 8); reading = reading | data_RX[1];
  sprintf(hexString, "%04X", reading);
  data_reading = round((float)reading*100)/10000.0;
  gateway_doc_p["Value"] = data_reading;
  node_doc["npk_p"] = data_reading;
  Serial.print("Phosphorous: ");
  Serial.print(data_reading);
  Serial.print(" mg/L\n");

  data_RX[0] = frameRX[7]; data_RX[1] = frameRX[8];
  reading = 0xFFFF & (data_RX[0] << 8); reading = reading | data_RX[1];
  sprintf(hexString, "%04X", reading);
```

```
      data_reading = round((float)reading*100)/10000.0;
      gateway_doc_k["Value"] = data_reading;
      node_doc["npk_k"] = data_reading;
      Serial.print("Potassium:␣");
      Serial.print(data_reading);
      Serial.print("␣mg/L\n");

    }
    Serial.println();
    digitalWrite(RE_DE_PIN, HIGH);

    vTaskDelay(pdMS_TO_TICKS(1000)); // Delay between transmissions
}

void initialize_rain(void)
{
    gateway_doc_rain["site_id"] = "RSSA_Site_01";
    gateway_doc_rain["device_id"] = nullptr;
    gateway_doc_rain["lat"] = nullptr; gateway_doc_rain["lon"] = nullptr;
    gateway_doc_rain["year"] = nullptr;
    gateway_doc_rain["doy"] = nullptr;
    gateway_doc_rain["time"] = "00:00:00";
    gateway_doc_rain["Quantity"] = "Rainfall";
    gateway_doc_rain["Value"] = nullptr;
    gateway_doc_rain["Unit"] = "mm";
}
void acquire_rain(uint8_t slave_id, const char* site_id, const char* device_id, float
    lat, float lon, uint8_t year, uint8_t doy, uint8_t hh, uint8_t mm, uint8_t ss)
{
    gateway_doc_rain["site_id"] = site_id;
    gateway_doc_rain["device_id"] = device_id;
    gateway_doc_rain["lat"] = lat; gateway_doc_rain["lon"] = lon;
    gateway_doc_rain["year"] = year;
    gateway_doc_rain["doy"] = doy;

    char timeStr[9]; // "hh:mm:ss" + null terminator
    sprintf(timeStr, "%02d:%02d:%02d", hh, mm, ss);
    gateway_doc_rain["time"] = timeStr;

    uint8_t frameData[] = {slave_id, 0x03, 0x00, 0x00, 0x00, 0x01, 0x00, 0x00}; //
    Slave address, Function code, Starting address, Number of registers, CRC Low, CRC
    High
    uint8_t slaveID_RX, opcode_RX, numBytes_RX, data_RX[2];
    uint16_t reading;
    float data_reading;

    node_doc["rain_sensor_address"] = slave_id;

    // Calculate CRC
    uint16_t crc = calculateCRC(frameData, sizeof(frameData) - 2); // Exclude CRC
    bytes from calculation

    // Add CRC to frame data
    frameData[sizeof(frameData) - 2] = crc & 0xFF; // Low byte of CRC
    frameData[sizeof(frameData) - 1] = (crc >> 8) & 0xFF; // High byte of CRC

    // Enable transmission
    digitalWrite(RE_DE_PIN, HIGH);
```

```
    // Send frame over SoftwareSerial
    for (uint8_t i = 0; i < sizeof(frameData); i++) {
        mySerial.write(frameData[i]);
    }

    // Disable transmission
    digitalWrite(RE_DE_PIN, LOW);

    // Print received bytes
    delay(500); // Wait for response
    uint8_t frameSize_RX = 7, i=0;
    uint8_t incomingByte;
    uint8_t frameRX[BUF_SIZE];
    while (mySerial.available()) {
        incomingByte = mySerial.read();
        frameRX[i] = incomingByte;
        i++;

    }
    char hexString[5];
    if(i){
      slaveID_RX = frameRX[0]; opcode_RX = frameRX[1]; numBytes_RX = frameRX[2];
      data_RX[0] = frameRX[3]; data_RX[1] = frameRX[4];
      reading = 0xFFFF & (data_RX[0] << 8); reading = reading | data_RX[1];
      sprintf(hexString, "%04X", reading);
      data_reading = round((float)reading*10)/100.0;
      gateway_doc_rain["Value"] = data_reading;
      node_doc["rain"] = data_reading;
      Serial.print("Rainfall␣Intensity:␣"); Serial.print(data_reading); Serial.print("
    ␣mm\n");

    }
    Serial.println("␣");
    digitalWrite(RE_DE_PIN, HIGH);

    vTaskDelay(pdMS_TO_TICKS(1000));
}

void initialize_light(void)
{
    gateway_doc_light["site_id"] = "RSSA_Site_01";
    gateway_doc_light["device_id"] = nullptr;
    gateway_doc_light["lat"] = nullptr; gateway_doc_light["lon"] = nullptr;
    gateway_doc_light["year"] = nullptr;
    gateway_doc_light["doy"] = nullptr;
    gateway_doc_light["time"] = "00:00:00";
    gateway_doc_light["Quantity"] = "Light";
    gateway_doc_light["Value"] = nullptr;
    gateway_doc_light["Unit"] = "Lux";
}
void acquire_light(uint8_t slave_id, const char* site_id, const char* device_id, float
     lat, float lon, uint8_t year, uint8_t doy, uint8_t hh, uint8_t mm, uint8_t ss)
{
    gateway_doc_light["site_id"] = site_id;
    gateway_doc_light["device_id"] = device_id;
    gateway_doc_light["lat"] = lat; gateway_doc_light["lon"] = lon;
    gateway_doc_light["year"] = year;
```

```
    gateway_doc_light["doy"] = doy;

    char timeStr[9]; // "hh:mm:ss" + null terminator
    sprintf(timeStr, "%02d:%02d:%02d", hh, mm, ss);
    gateway_doc_light["time"] = timeStr;

    uint8_t frameData[] = {slave_id, 0x03, 0x00, 0x00, 0x00, 0x01, 0x00, 0x00}; //
    Slave address, Function code, Starting address, Number of registers, CRC Low, CRC
    High
    uint8_t slaveID_RX, opcode_RX, numBytes_RX, data_RX[2];
    uint16_t reading;
    float data_reading;

    node_doc["light_sensor_address"] = slave_id;

    // Calculate CRC
    uint16_t crc = calculateCRC(frameData, sizeof(frameData) - 2); // Exclude CRC
    bytes from calculation

    // Add CRC to frame data
    frameData[sizeof(frameData) - 2] = crc & 0xFF; // Low byte of CRC
    frameData[sizeof(frameData) - 1] = (crc >> 8) & 0xFF; // High byte of CRC

    // Enable transmission
    digitalWrite(RE_DE_PIN, HIGH);

    // Send frame over SoftwareSerial
    for (uint8_t i = 0; i < sizeof(frameData); i++) {
        mySerial.write(frameData[i]);
    }

    // Disable transmission
    digitalWrite(RE_DE_PIN, LOW);

    // Print received bytes
    delay(500); // Wait for response
    uint8_t frameSize_RX = 7, i=0;
    uint8_t incomingByte;
    uint8_t frameRX[BUF_SIZE];
    while (mySerial.available()) {
        incomingByte = mySerial.read();
        frameRX[i] = incomingByte;
        i++;

    }
    char hexString[5];
    if(i){
      slaveID_RX = frameRX[0]; opcode_RX = frameRX[1]; numBytes_RX = frameRX[2];
      data_RX[0] = frameRX[3]; data_RX[1] = frameRX[4];
      reading = 0xFFFF & (data_RX[0] << 8); reading = reading | data_RX[1];
      sprintf(hexString, "%04X", reading);
      data_reading = round((float)reading*10)/100.0;
      gateway_doc_light["Value"] = data_reading;
      node_doc["light"] = data_reading;
      Serial.print("Rainfall␣Intensity:␣"); Serial.print(data_reading); Serial.print("
    ␣mm\n");

    }
```

```
    Serial.println("␣");
    digitalWrite(RE_DE_PIN, HIGH);

    vTaskDelay(pdMS_TO_TICKS(1000));
}

void initialize_soil_seeed(void)
{
    gateway_doc_soil_temp_seeed["site_id"] = "RSSA_Site_01";
    gateway_doc_soil_temp_seeed["device_id"] = nullptr;
    gateway_doc_soil_temp_seeed["lat"] = nullptr; gateway_doc_soil_temp_seeed["lon"] =
     nullptr;
    gateway_doc_soil_temp_seeed["year"] = nullptr;
    gateway_doc_soil_temp_seeed["doy"] = nullptr;
    gateway_doc_soil_temp_seeed["time"] = "00:00:00";
    gateway_doc_soil_temp_seeed["Quantity"] = "Soil␣Temp";
    gateway_doc_soil_temp_seeed["Value"] = nullptr;
    gateway_doc_soil_temp_seeed["Unit"] = "Celsius";

    gateway_doc_soil_moist_seeed["site_id"] = "RSSA_Site_01";
    gateway_doc_soil_moist_seeed["device_id"] = nullptr;
    gateway_doc_soil_moist_seeed["lat"] = nullptr; gateway_doc_soil_moist_seeed["lon"]
     = nullptr;
    gateway_doc_soil_moist_seeed["year"] = nullptr;
    gateway_doc_soil_moist_seeed["doy"] = nullptr;
    gateway_doc_soil_moist_seeed["time"] = "00:00:00";
    gateway_doc_soil_moist_seeed["Quantity"] = "Soil␣Moisture";
    gateway_doc_soil_moist_seeed["Value"] = nullptr;
    gateway_doc_soil_moist_seeed["Unit"] = "%VWC";

    gateway_doc_soil_ec_seeed["site_id"] = "RSSA_Site_01";
    gateway_doc_soil_ec_seeed["device_id"] = nullptr;
    gateway_doc_soil_ec_seeed["lat"] = nullptr; gateway_doc_soil_ec_seeed["lon"] =
    nullptr;
    gateway_doc_soil_ec_seeed["year"] = nullptr;
    gateway_doc_soil_ec_seeed["doy"] = nullptr;
    gateway_doc_soil_ec_seeed["time"] = "00:00:00";
    gateway_doc_soil_ec_seeed["Quantity"] = "Soil␣EC";
    gateway_doc_soil_ec_seeed["Value"] = nullptr;
    gateway_doc_soil_ec_seeed["Unit"] = "us/cm";
}
void acquire_soil_seeed(uint8_t slave_id, const char* site_id, const char* device_id,
    float lat, float lon, uint8_t year, uint8_t doy, uint8_t hh, uint8_t mm, uint8_t
    ss)
{
    gateway_doc_soil_temp_seeed["site_id"] = site_id; gateway_doc_soil_moist_seeed["
    site_id"] = site_id; gateway_doc_soil_ec_seeed["site_id"] = site_id;
    gateway_doc_soil_temp_seeed["device_id"] = device_id; gateway_doc_soil_moist_seeed
    ["device_id"] = device_id; gateway_doc_soil_ec_seeed["device_id"] = device_id;
    gateway_doc_soil_temp_seeed["lat"] = lat; gateway_doc_soil_temp_seeed["lon"] = lon;
     gateway_doc_soil_moist_seeed["lat"] = lat; gateway_doc_soil_moist_seeed["lon"] =
    lon; gateway_doc_soil_ec_seeed["lat"] = lat; gateway_doc_soil_ec_seeed["lon"] =
    lon;
    gateway_doc_soil_temp_seeed["year"] = year; gateway_doc_soil_moist_seeed["year"] =
     year; gateway_doc_soil_ec_seeed["year"] = year;
    gateway_doc_soil_temp_seeed["doy"] = doy; gateway_doc_soil_moist_seeed["doy"] =
    doy; gateway_doc_soil_ec_seeed["doy"] = doy;
```

```
    char timeStr[9]; // "hh:mm:ss" + null terminator
    sprintf(timeStr, "%02d:%02d:%02d", hh, mm, ss);
    gateway_doc_soil_temp_seeed["time"] = timeStr; gateway_doc_soil_moist_seeed["time"
    ] = timeStr; gateway_doc_soil_ec_seeed["time"] = timeStr;

    uint8_t frameData[] = {slave_id, 0x04, 0x00, 0x00, 0x00, 0x03, 0x00, 0x00}; //
    Slave address, Function code, Starting address, Number of registers, CRC Low, CRC
    High
    uint8_t slaveID_RX, opcode_RX, numBytes_RX, data_RX[2];
    uint16_t reading;
    float data_reading;

    node_doc["soil_temp_moisture_ec_sensor"] = "Seeed_TEM_01";
    node_doc["soil_temp_moisture_ec_sensor_address"] = slave_id;

    // Calculate CRC
    uint16_t crc = calculateCRC(frameData, sizeof(frameData) - 2); // Exclude CRC
    bytes from calculation

    // Add CRC to frame data
    frameData[sizeof(frameData) - 2] = crc & 0xFF; // Low byte of CRC
    frameData[sizeof(frameData) - 1] = (crc >> 8) & 0xFF; // High byte of CRC

    // Enable transmission
    digitalWrite(RE_DE_PIN, HIGH);

    // Send frame over SoftwareSerial
    for (uint8_t i = 0; i < sizeof(frameData); i++) {
        mySerial.write(frameData[i]);
    }

    // Disable transmission
    digitalWrite(RE_DE_PIN, LOW);

    // Print received bytes
    delay(500); // Wait for response
    uint8_t frameSize_RX = 7, i=0;
    uint8_t incomingByte;
    uint8_t frameRX[BUF_SIZE];
    while (mySerial.available()) {
        incomingByte = mySerial.read();
        frameRX[i] = incomingByte;
        i++;

    }
    slaveID_RX = frameRX[0]; opcode_RX = frameRX[1]; numBytes_RX = frameRX[2];
    data_RX[0] = frameRX[3]; data_RX[1] = frameRX[4];
    reading = 0xFFFF & (data_RX[0] << 8); reading = reading | data_RX[1];
    data_reading = round((float)reading*100)/10000.0;
    gateway_doc_soil_temp_seeed["Value"] = data_reading;
    node_doc["soil_temp"] = data_reading;
    Serial.print("Soil␣Temperature:␣"); Serial.print(data_reading); Serial.print("␣
    Celsius\n");

    data_RX[0] = frameRX[5]; data_RX[1] = frameRX[6];
    reading = 0xFFFF & (data_RX[0] << 8); reading = reading | data_RX[1];
    data_reading = round((float)reading*100)/10000.0;
    gateway_doc_soil_moist_seeed["Value"] = data_reading;
```

```
    node_doc["soil_moisture"] = data_reading;
    Serial.print("Soil␣Moisture:␣"); Serial.print(data_reading); Serial.print("␣%RH\n"
    );

    data_RX[0] = frameRX[7]; data_RX[1] = frameRX[8];
    reading = 0xFFFF & (data_RX[0] << 8); reading = reading | data_RX[1];
    data_reading = round((float)reading*100)/100.0;
    gateway_doc_soil_ec_seeed["Value"] = data_reading;
    node_doc["soil_ec"] = data_reading;
    Serial.print("Soil␣EC:␣"); Serial.print(data_reading); Serial.print("␣uS/cm\n");
    Serial.println("␣");
    digitalWrite(RE_DE_PIN, HIGH);

    vTaskDelay(pdMS_TO_TICKS(1000));
}

void initialize_soil_niubol(void)
{
    gateway_doc_soil_temp_niubol["site_id"] = "RSSA_Site_01";
    gateway_doc_soil_temp_niubol["device_id"] = nullptr;
    gateway_doc_soil_temp_niubol["lat"] = nullptr; gateway_doc_soil_temp_niubol["lon"]
     = nullptr;
    gateway_doc_soil_temp_niubol["year"] = nullptr;
    gateway_doc_soil_temp_niubol["doy"] = nullptr;
    gateway_doc_soil_temp_niubol["time"] = "00:00:00";
    gateway_doc_soil_temp_niubol["Quantity"] = "Soil␣Temp";
    gateway_doc_soil_temp_niubol["Value"] = nullptr;
    gateway_doc_soil_temp_niubol["Unit"] = "Celsius";

    gateway_doc_soil_moist_niubol["site_id"] = "RSSA_Site_01";
    gateway_doc_soil_moist_niubol["device_id"] = nullptr;
    gateway_doc_soil_moist_niubol["lat"] = nullptr; gateway_doc_soil_moist_niubol["lon"
    ] = nullptr;
    gateway_doc_soil_moist_niubol["year"] = nullptr;
    gateway_doc_soil_moist_niubol["doy"] = nullptr;
    gateway_doc_soil_moist_niubol["time"] = "00:00:00";
    gateway_doc_soil_moist_niubol["Quantity"] = "Soil␣Moisture";
    gateway_doc_soil_moist_niubol["Value"] = nullptr;
    gateway_doc_soil_moist_niubol["Unit"] = "%VWC";

    gateway_doc_soil_ec_niubol["site_id"] = "RSSA_Site_01";
    gateway_doc_soil_ec_niubol["device_id"] = nullptr;
    gateway_doc_soil_ec_niubol["lat"] = nullptr; gateway_doc_soil_ec_niubol["lon"] =
    nullptr;
    gateway_doc_soil_ec_niubol["year"] = nullptr;
    gateway_doc_soil_ec_niubol["doy"] = nullptr;
    gateway_doc_soil_ec_niubol["time"] = "00:00:00";
    gateway_doc_soil_ec_niubol["Quantity"] = "Soil␣EC";
    gateway_doc_soil_ec_niubol["Value"] = nullptr;
    gateway_doc_soil_ec_niubol["Unit"] = "us/cm";
}
void acquire_soil_niubol(uint8_t slave_id, const char* site_id, const char* device_id,
     float lat, float lon, uint8_t year, uint8_t doy, uint8_t hh, uint8_t mm, uint8_t
    ss)
{
    gateway_doc_soil_temp_niubol["site_id"] = site_id; gateway_doc_soil_moist_niubol["
    site_id"] = site_id; gateway_doc_soil_ec_niubol["site_id"] = site_id;
    gateway_doc_soil_temp_niubol["device_id"] = device_id;
```

```
gateway_doc_soil_moist_niubol["device_id"] = device_id; gateway_doc_soil_ec_niubol
["device_id"] = device_id;
gateway_doc_soil_temp_niubol["lat"] = lat; gateway_doc_soil_temp_niubol["lon"] =
lon; gateway_doc_soil_moist_niubol["lat"] = lat; gateway_doc_soil_moist_niubol["
lon"] = lon; gateway_doc_soil_ec_niubol["lat"] = lat; gateway_doc_soil_ec_niubol["
lon"] = lon;
gateway_doc_soil_temp_niubol["year"] = year; gateway_doc_soil_moist_niubol["year"]
 = year; gateway_doc_soil_ec_niubol["year"] = year;
gateway_doc_soil_temp_niubol["doy"] = doy; gateway_doc_soil_moist_niubol["doy"] =
doy; gateway_doc_soil_ec_niubol["doy"] = doy;

char timeStr[9]; // "hh:mm:ss" + null terminator
sprintf(timeStr, "%02d:%02d:%02d", hh, mm, ss);
gateway_doc_soil_temp_niubol["time"] = timeStr; gateway_doc_soil_moist_niubol["
time"] = timeStr; gateway_doc_soil_ec_niubol["time"] = timeStr;

uint8_t frameData[] = {slave_id, 0x04, 0x00, 0x00, 0x00, 0x03, 0x00, 0x00}; //
Slave address, Function code, Starting address, Number of registers, CRC Low, CRC
High
uint8_t slaveID_RX, opcode_RX, numBytes_RX, data_RX[2];
uint16_t reading;
float data_reading;

node_doc["soil_temp_moisture_ec_sensor"] = "NiuBol_DKI_TEM_01";
node_doc["soil_temp_moisture_ec_sensor_address"] = slave_id;

// Calculate CRC
uint16_t crc = calculateCRC(frameData, sizeof(frameData) - 2); // Exclude CRC
bytes from calculation

// Add CRC to frame data
frameData[sizeof(frameData) - 2] = crc & 0xFF; // Low byte of CRC
frameData[sizeof(frameData) - 1] = (crc >> 8) & 0xFF; // High byte of CRC

// Enable transmission
digitalWrite(RE_DE_PIN, HIGH);

// Send frame over SoftwareSerial
for (uint8_t i = 0; i < sizeof(frameData); i++) {
    mySerial.write(frameData[i]);
}

// Disable transmission
digitalWrite(RE_DE_PIN, LOW);

// Print received bytes
delay(500); // Wait for response
uint8_t frameSize_RX = 7, i=0;
uint8_t incomingByte;
uint8_t frameRX[BUF_SIZE];
while (mySerial.available()) {
    incomingByte = mySerial.read();
    frameRX[i] = incomingByte;
    i++;

}
slaveID_RX = frameRX[0]; opcode_RX = frameRX[1]; numBytes_RX = frameRX[2];
data_RX[0] = frameRX[3]; data_RX[1] = frameRX[4];
```

```
    reading = 0xFFFF & (data_RX[0] << 8); reading = reading | data_RX[1];
    data_reading = round((float)reading*100)/10000.0;
    gateway_doc_soil_temp_niubol["Value"] = data_reading;
    node_doc["soil_temp"] = data_reading;
    Serial.print("Soil␣Temperature:␣"); Serial.print(data_reading); Serial.print("␣
    Celsius\n");

    data_RX[0] = frameRX[5]; data_RX[1] = frameRX[6];
    reading = 0xFFFF & (data_RX[0] << 8); reading = reading | data_RX[1];
    data_reading = round((float)reading*100)/10000.0;
    gateway_doc_soil_moist_niubol["Value"] = data_reading;
    node_doc["soil_moisture"] = data_reading;
    Serial.print("Soil␣Moisture:␣"); Serial.print(data_reading); Serial.print("␣%RH\n"
    );

    data_RX[0] = frameRX[7]; data_RX[1] = frameRX[8];
    reading = 0xFFFF & (data_RX[0] << 8); reading = reading | data_RX[1];
    data_reading = round((float)reading*100)/100.0;
    gateway_doc_soil_ec_niubol["Value"] = data_reading;
    node_doc["soil_ec"] = data_reading;
    Serial.print("Soil␣EC:␣"); Serial.print(data_reading); Serial.print("␣uS/cm\n");
    Serial.println("␣");
    digitalWrite(RE_DE_PIN, HIGH);

    vTaskDelay(pdMS_TO_TICKS(1000));
}

void initialize_leaf(void)
{
    gateway_doc_leaf_temp["site_id"] = "RSSA_Site_01";
    gateway_doc_leaf_temp["device_id"] = nullptr;
    gateway_doc_leaf_temp["lat"] = nullptr; gateway_doc_leaf_temp["lon"] = nullptr;
    gateway_doc_leaf_temp["year"] = nullptr;
    gateway_doc_leaf_temp["doy"] = nullptr;
    gateway_doc_leaf_temp["time"] = "00:00:00";
    gateway_doc_leaf_temp["Quantity"] = "Leaf␣Temp";
    gateway_doc_leaf_temp["Value"] = nullptr;
    gateway_doc_leaf_temp["Unit"] = "Celsius";

    gateway_doc_leaf_wet["site_id"] = "RSSA_Site_01";
    gateway_doc_leaf_wet["device_id"] = nullptr;
    gateway_doc_leaf_wet["lat"] = nullptr; gateway_doc_leaf_wet["lon"] = nullptr;
    gateway_doc_leaf_wet["year"] = nullptr;
    gateway_doc_leaf_wet["doy"] = nullptr;
    gateway_doc_leaf_wet["time"] = "00:00:00";
    gateway_doc_leaf_wet["Quantity"] = "Leaf␣Wetness";
    gateway_doc_leaf_wet["Value"] = nullptr;
    gateway_doc_leaf_wet["Unit"] = "%RH";
}
void acquire_leaf(uint8_t slave_id, const char* site_id, const char* device_id, float
    lat, float lon, uint8_t year, uint8_t doy, uint8_t hh, uint8_t mm, uint8_t ss)
{
    gateway_doc_leaf_temp["site_id"] = site_id; gateway_doc_leaf_wet["site_id"] =
    site_id;
    gateway_doc_leaf_temp["device_id"] = device_id; gateway_doc_leaf_wet["device_id"]
    = device_id;
    gateway_doc_leaf_temp["lat"] = lat; gateway_doc_leaf_temp["lon"] = lon;
    gateway_doc_leaf_wet["lat"] = lat; gateway_doc_leaf_wet["lon"] = lon;
```

```
gateway_doc_leaf_temp["year"] = year; gateway_doc_leaf_wet["year"] = year;
gateway_doc_leaf_temp["doy"] = doy; gateway_doc_leaf_wet["doy"] = doy;

char timeStr[9]; // "hh:mm:ss" + null terminator
sprintf(timeStr, "%02d:%02d:%02d", hh, mm, ss);
gateway_doc_leaf_temp["time"] = timeStr; gateway_doc_leaf_wet["time"] = timeStr;


uint8_t frameData[] = {0x07, 0x03, 0x00, 0x00, 0x00, 0x02, 0x00, 0x00}; // Slave
address, Function code, Starting address, Number of registers, CRC Low, CRC High
uint8_t slaveID_RX, opcode_RX, numBytes_RX, data_RX[2];
uint16_t reading;
float data_reading;

node_doc["leaf_sensor_address"] = slave_id;

// Calculate CRC
uint16_t crc = calculateCRC(frameData, sizeof(frameData) - 2); // Exclude CRC
bytes from calculation

// Add CRC to frame data
frameData[sizeof(frameData) - 2] = crc & 0xFF; // Low byte of CRC
frameData[sizeof(frameData) - 1] = (crc >> 8) & 0xFF; // High byte of CRC

// Enable transmission
digitalWrite(RE_DE_PIN, HIGH);

// Send frame over SoftwareSerial
for (uint8_t i = 0; i < sizeof(frameData); i++) {
    mySerial.write(frameData[i]);
}

// Disable transmission
digitalWrite(RE_DE_PIN, LOW);

// Print received bytes
delay(500); // Wait for response
uint8_t frameSize_RX = 7, i=0;
uint8_t incomingByte;
uint8_t frameRX[BUF_SIZE];
while (mySerial.available()) {
    incomingByte = mySerial.read();
    frameRX[i] = incomingByte;
    i++;

}
char hexString[5];
if(i){

  slaveID_RX = frameRX[0]; opcode_RX = frameRX[1]; numBytes_RX = frameRX[2];

  data_RX[0] = frameRX[3]; data_RX[1] = frameRX[4];
  reading = 0xFFFF & (data_RX[0] << 8); reading = reading | data_RX[1];
  sprintf(hexString, "%04X", reading);
  data_reading = round((float)reading*100)/10000.0;
  Serial.print("Leaf␣Temperature:␣"); Serial.print(data_reading); Serial.print("␣
Celsius\n");
  node_doc["leaf_temp"] = data_reading;
```

```
      gateway_doc_leaf_temp["Value"] = data_reading;

      data_RX[0] = frameRX[5]; data_RX[1] = frameRX[6];
      reading = 0xFFFF & (data_RX[0] << 8); reading = reading | data_RX[1];
      sprintf(hexString, "%04X", reading);
      data_reading = round((float)reading*10)/100.0;
      Serial.print("Leaf␣Wetness:␣"); Serial.print(data_reading); Serial.print("␣%RH\n"
    );
      node_doc["leaf_wet"] = data_reading;
      gateway_doc_leaf_wet["Value"] = data_reading;
    }
    Serial.println("␣");
    digitalWrite(RE_DE_PIN, HIGH);

    vTaskDelay(pdMS_TO_TICKS(1000));
}
```

The following code block is the main code that uses these custom header files and performs the requisite operations of the node:

```
/*
 * RSSA Node Software
 *
 * Author:
 *   Swastik Bhattacharya
 *
 * Date:
 *   July 30, 2024
 */
// Enabling Sensors
// #define PH_ENABLED
#define NPK_ENABLED
// #define RAIN_ENABLED
// #define LIGHT_ENABLED
// #define SOIL_SEEED_ENABLED
// #define SOIL_NIUBOL_ENABLED
// #define LEAF_ENABLED
// #define GPS_ENABLED
#define GPS_NOT_ENABLED

#include "RSSAModbusJSONLib.h"
#include <Arduino.h>
#include <SoftwareSerial.h>
#include <HardwareSerial.h>
#include <TinyGPS++.h>
#include <SPI.h>
#include <SD.h>
#include <ArduinoJson.h>
// #include <LoRa.h>
#include <WiFi.h>
#include <PubSubClient.h>
// #include <RH_RF95.h>
#include <U8x8lib.h>

#ifdef U8X8_HAVE_HW_SPI
#include <SPI.h>
#endif

#define TXD_PIN 26
```

```
#define RXD_PIN 27
#define RE_DE_PIN 25
#define BUF_SIZE 1024

#define GPS_TX_PIN 16
#define GPS_RX_PIN 17

#define SD_CS_PIN 5    //SD Card Chip Select

/*---------Radio Setup-----------*/
#define Tx_Address 1
#define Rx_Address 2

#define RST_Pin 13
#define CSS_Pin 33
#define Int_Pin 12
#define Data_Pin 25

// Change to 434.0 or other frequency, must match RX's freq!
// RF95: 865MHz   RF96: 433MHz  915   434
#define Tx_Freq 915.0

#define MAX_JSON_CHUNK_LEN 100 // Maximum length of each JSON chunk

#define RING_BUFFER_SIZE 30  // Define the size of the ring buffer
#define JSON_DOCUMENT_SIZE 2048  // Adjust the size of each JSON document

// Defining Slave IDs
#define PH_SLAVE_ID 1
#define NPK_SLAVE_ID 2
#define RAIN_SLAVE_ID 3
#define LIGHT_SLAVE_ID 4
#define SOIL_SEEED_SLAVE_ID 5
#define SOIL_NIUBOL_SLAVE_ID 6
#define LEAF_SLAVE_ID 7

U8X8_SSD1306_128X64_NONAME_SW_I2C u8x8(/* clock=*/ SCL, /* data=*/ SDA, /* reset=*/
    U8X8_PIN_NONE);   // OLEDs without Reset of the Display

// Defining Site ID

const char* site_id = "RSSA_BLR_01";

// Defining Device IDs

const char *ph_device_id = "NiuBol_DKI_PH_01";
const char *npk_device_id = "NiuBol_DKI_NPK_01";
const char *rain_device_id = "NiuBol_DKI_Rain_01";
const char *light_device_id = "NiuBol_DKI_Light_01";
const char *soil_seeed_device_id = "Seeed_TEM_01";
const char *soil_niubol_device_id = "NiuBol_DKI_TEM_01";
const char *leaf_device_id = "NiuBol_DKI_Leaf_01";

// WiFi settings
const char *ssid = "RSSA_AP";     // SSID of the Raspberry Pi AP
const char *password = "RSSA_12345678"; // Password for the Raspberry Pi

// MQTT settings
```

```
const char *mqtt_server = "192.168.4.1"; // IP address of the Raspberry Pi
const char *mqtt_topic_pH = "pH_data";
const char *mqtt_topic_n = "n_data";
const char *mqtt_topic_p = "p_data";
const char *mqtt_topic_k = "k_data";
const char *mqtt_topic_rain = "rain_data";
const char *mqtt_topic_light = "light_data";
const char *mqtt_topic_s_temp = "s_temp_data";
const char *mqtt_topic_s_moist = "s_moist_data";
const char *mqtt_topic_s_ec = "s_ec_data";
const char *mqtt_topic_l_wet = "l_wet_data";
const char *mqtt_topic_l_temp = "l_temp_data";
const char* mqtt_username = "rssa-gateway"; // MQTT username
const char* mqtt_password = "rssa_gateway"; // MQTT password
const char* clientID = "RSSA_Node"; // MQTT client ID

WiFiClient espClient;
PubSubClient client(mqtt_server, 1883, espClient);

// Singleton instance of the radio driver
// RH_RF95 rf95(CSS_Pin, Int_Pin);

SemaphoreHandle_t modbusSemaphore; // Semaphore for triggering the modbusTask
TinyGPSPlus gps; // Define the gps object
String filenameString;

#define GPS_PORT Serial2

int year, month, day, hour, minute, second, doy;

unsigned long startTime, elapsedTime;

char buffer[20];

void extractGPSInfoFileName() {
  // Variables to store GPS data

  unsigned long fix_age;

    if (gps.location.isValid() && gps.date.isValid() && gps.time.isValid()) {
    // Extract date and time components
    year = gps.date.year();
    month = gps.date.month();
    day = gps.date.day();
    hour = gps.time.hour();
    minute = gps.time.minute();
    second = gps.time.second();
    }
  // Extract date and time
  // gps.crack_datetime(&year, &month, &day, &hour, &minute, &second, &fix_age);

  // Convert day, hour, minute, and second to day of year
  doy = dayOfYear(year, month, day);

  node_doc["year"] = year;
  node_doc["doy"] = doy;
  node_doc["hh"] = hour;
  node_doc["mm"] = minute;
```

```
  node_doc["ss"] = second;

  node_doc["lat"] = round(gps.location.lat()*100)/100.0; node_doc["lon"] = round(gps.
    location.lng()*100)/100.0;

  // Create a string with the extracted data
  filenameString = "/" + String(site_id) + "_" + String(year) + "_" + String(doy) + ".
    json";

  // Print the timestamp string
  Serial.println("filename:␣" + filenameString);
}

// Function to calculate day of year
int dayOfYear(int year, int month, int day) {
  static int days[] = {0,31,59,90,120,151,181,212,243,273,304,334};
  int doy = days[month - 1] + day;
  if (month > 2 && year % 4 == 0 && (year % 100 != 0 || year % 400 == 0))
    doy++;
  return doy;
}

void createNewFile(const char* filename) {
  // Open the file in write mode
  File file = SD.open(filename, FILE_WRITE);

  // If the file opened successfully
  if (file) {
    // Close the file
    file.close();
    Serial.println("File␣created␣successfully.");
  } else {
    Serial.println("Error␣creating␣file.");
  }
}

void appendToFile(const char* filename, String data) {
  // Open the file in append mode
  File file = SD.open(filename, FILE_APPEND);

  // If the file opened successfully
  if (file) {
    // Append data to the file
    file.print(data);

    // Close the file
    file.close();
  } else {
    Serial.println("Error␣appending␣to␣file.");
  }
}

void reconnect() {
  // Loop until we're reconnected
  int attempts = 0;
  while (!client.connected()) {
    Serial.print("Attempting␣MQTT␣connection...");
    // Attempt to connect
```

```
    if (client.connect(clientID, mqtt_username, mqtt_password)) {
      Serial.println("connected");
    } else {
      Serial.print("failed, rc=");
      Serial.print(client.state());
      Serial.println(" try again in 5 seconds");
      attempts++;
      // Wait 5 seconds before retrying
      delay(5000);
    }
    if(attempts >= 5)
      break;
  }
}

void setup() {
    year = 2024; hour = 0; minute=0; second=0; doy=1;
    Serial.begin(9600);
    mySerial.begin(9600);
    GPS_PORT.begin(9600);
    pinMode(RE_DE_PIN, OUTPUT);
    digitalWrite(RE_DE_PIN, LOW);
    Serial.println("Setup Complete");

    if (!SD.begin(SD_CS_PIN)) {
      Serial.println("SD card initialization failed!");
        // return;
    }
  Serial.println("SD card initialized.");

  u8x8.begin();
  u8x8.setPowerSave(0);

  u8x8.setFont(u8x8_font_chroma48medium8_r);

  startTime = millis();

  initialize_node_json();

  #ifdef PH_ENABLED
  initialize_pH();
  u8x8.clearDisplay();
  u8x8.drawString(1,1,"pH defined...");
  delay(2000);
  #endif

  #ifdef NPK_ENABLED
  initialize_npk();
  u8x8.clearDisplay();
  u8x8.drawString(1,1,"NPK defined...");
  delay(2000);
  #endif

  #ifdef RAIN_ENABLED
  initialize_rain();
  u8x8.clearDisplay();
  u8x8.drawString(1,1,"Rainfall defined...");
  delay(2000);
```

```
  #endif

  #ifdef LIGHT_ENABLED
  initialize_light();
  u8x8.clearDisplay();
  u8x8.drawString(1,1,"Light␣defined...");
  delay(2000);
  #endif

  #ifdef SOIL_SEEED_ENABLED
  initialize_soil_seeed();
  u8x8.clearDisplay();
  u8x8.drawString(1,1,"Soil␣TEM␣defined...");
  delay(2000);
  #endif

  #ifdef SOIL_NIUBOL_ENABLED
  initialize_soil_niubol();
  u8x8.clearDisplay();
  u8x8.drawString(1,1,"Soil␣TEM␣defined...");
  delay(2000);
  #endif

  #ifdef LEAF_ENABLED
  initialize_leaf();
  u8x8.clearDisplay();
  u8x8.drawString(1,1,"Leaf␣defined...");
  delay(2000);
  #endif


// Wait for a valid GPS fix

#ifdef GPS_ENABLED
u8x8.clearDisplay();
while(1)
{
  while (GPS_PORT.available() > 0) {
    gps.encode(GPS_PORT.read());
  }
  if (gps.location.isValid()) {
    // Print latitude and longitude
    u8x8.clearDisplay();
    u8x8.setInverseFont(1);
    u8x8.drawString(1, 0, "GPS␣FIX");
    u8x8.setInverseFont(0);
    Serial.print("Latitude:␣");
//    u8x8.drawString(1, 1, "Lat: " + String(gps.location.lat()));
    snprintf(buffer, sizeof(buffer), "Lat:␣%.4f", gps.location.lat());
    u8x8.drawString(1, 1, buffer);
    Serial.print(gps.location.lat(), 6);
    Serial.print(",␣Longitude:␣");
    snprintf(buffer, sizeof(buffer), "Lon:␣%.4f", gps.location.lng());
    u8x8.drawString(1, 2, buffer);
    Serial.println(gps.location.lng(), 6);

    // Print date and time (if available)
    if (gps.date.isValid() && gps.time.isValid()) {
```

```
      Serial.print("Date: ");
      Serial.print(gps.date.month());
      Serial.print("/");
      Serial.print(gps.date.day());
      Serial.print("/");
      Serial.print(gps.date.year());
      Serial.print(" Time: ");
      if (gps.time.hour() < 10) Serial.print('0');
      Serial.print(gps.time.hour());
      Serial.print(":");
      if (gps.time.minute() < 10) Serial.print('0');
      Serial.print(gps.time.minute());
      Serial.print(":");
      if (gps.time.second() < 10) Serial.print('0');
      Serial.println(gps.time.second());
    }

    // Print number of satellites in view
    Serial.print("Satellites in view: ");
    Serial.println(gps.satellites.value());
    break;
  } else {
    u8x8.clearDisplay();
    Serial.println("Waiting for valid GPS fix...");
    u8x8.drawString(1, 1, "Waiting for GPS");
  }

  delay(1000); // Update every second
}
#endif


// Connect to WiFi
WiFi.begin(ssid, password);
while (WiFi.status() != WL_CONNECTED) {
    delay(500);
    Serial.println("Connecting to WiFi..");
    u8x8.clearDisplay();
    u8x8.drawString(1, 1, "Connecting to WiFi");
}
Serial.println("Connected to the WiFi network");
// Print local IP address and start web server
Serial.println("");
Serial.println("WiFi connected.");
u8x8.clearDisplay();
u8x8.drawString(1, 1, "WiFi connected.");
Serial.println("IP address: ");
Serial.println(WiFi.localIP());
  // Set up the MQTT client
  client.setServer(mqtt_server, 1883);
  reconnect();

pinMode(RST_Pin, OUTPUT);
  digitalWrite(RST_Pin, HIGH);

  delay(100);

  analogReadResolution(12);
```

```
  delay(100);

}

void loop() {


    //Extract file names
    #ifdef GPS_ENABLED
    extractGPSInfoFileName();
    #endif

    #ifdef GPS_NOT_ENABLED
    elapsedTime = millis() - startTime;

    // Convert milliseconds to days, hours, minutes, and seconds
    unsigned long seconds = elapsedTime / 1000;
    unsigned long minutes = seconds / 60;
    unsigned long hours = minutes / 60;
    unsigned long doy = hours / 24;

    // Calculate remaining hours, minutes, and seconds
    seconds %= 60;
    minutes %= 60;
    hours %= 24;

    node_doc["doy"] = doy;
    node_doc["hh"] = hours;
    node_doc["mm"] = minutes;
    node_doc["ss"] = seconds;
    #endif


    //Acquire pH
    #ifdef PH_ENABLED
    acquire_pH(PH_SLAVE_ID, site_id, ph_device_id, node_doc["lat"], node_doc["lat"],
    node_doc["year"], node_doc["doy"], node_doc["hh"], node_doc["mm"], node_doc["ss"]);

    u8x8.clearDisplay();
    u8x8.setInverseFont(1);
    u8x8.drawString(1,0,"Soil␣pH");
    u8x8.setInverseFont(0);
    snprintf(buffer, sizeof(buffer), "pH:␣%.4f", node_doc["ph"]);
    u8x8.drawString(1, 2, buffer);
    delay(20000);
    #endif

    //Acquire NPK
    #ifdef NPK_ENABLED
    acquire_npk(NPK_SLAVE_ID, site_id, npk_device_id, node_doc["lat"], node_doc["lat"],
     node_doc["year"], node_doc["doy"], node_doc["hh"], node_doc["mm"], node_doc["ss"])
    ;
    u8x8.clearDisplay();
    u8x8.setInverseFont(1);
    u8x8.drawString(1,0,"Soil␣NPK");
    u8x8.setInverseFont(0);
    snprintf(buffer, sizeof(buffer), "N:␣%.4f", node_doc["npk_n"]);
    u8x8.drawString(1, 2, buffer);
```

```
    snprintf(buffer, sizeof(buffer), "P:␣%.4f", node_doc["npk_p"]);
    u8x8.drawString(1, 3, buffer);
    snprintf(buffer, sizeof(buffer), "K:␣%.4f", node_doc["npk_k"]);
    u8x8.drawString(1, 4, buffer);
    delay(20000);
    #endif

    //Acquire Rain
    #ifdef RAIN_ENABLED
    acquire_rain(RAIN_SLAVE_ID, site_id, rain_device_id, node_doc["lat"], node_doc["
    lat"], node_doc["year"], node_doc["doy"], node_doc["hh"], node_doc["mm"], node_doc
    ["ss"]);
    u8x8.clearDisplay();
    u8x8.setInverseFont(1);
    u8x8.drawString(1,0,"Rain");
    u8x8.setInverseFont(0);
    snprintf(buffer, sizeof(buffer), "Rain:␣%.4f", node_doc["rain"]);
    u8x8.drawString(1, 2, buffer);
    delay(20000);
    #endif

    //Acquire Light
    #ifdef LIGHT_ENABLED
    acquire_light(LIGHT_SLAVE_ID, site_id, light_device_id, node_doc["lat"], node_doc["
    lat"], node_doc["year"], node_doc["doy"], node_doc["hh"], node_doc["mm"], node_doc
    ["ss"]);
    u8x8.clearDisplay();
    u8x8.setInverseFont(1);
    u8x8.drawString(1,0,"Light");
    u8x8.setInverseFont(0);
    snprintf(buffer, sizeof(buffer), "Light:␣%.4f␣", node_doc["light"]);
    u8x8.drawString(1, 2, buffer);
    delay(20000);
    #endif


    //Acquire TEM from SEEED
    #ifdef SOIL_SEEED_ENABLED
    acquire_soil_seeed(SOIL_SEEED_SLAVE_ID, site_id, soil_seeed_device_id, node_doc["
    lat"], node_doc["lat"], node_doc["year"], node_doc["doy"], node_doc["hh"],
    node_doc["mm"], node_doc["ss"]);
    u8x8.clearDisplay();
    u8x8.setInverseFont(1);
    u8x8.drawString(1,0,"Soil␣TEM");
    u8x8.setInverseFont(0);
    snprintf(buffer, sizeof(buffer), "Soil␣Temp:␣%.4f", node_doc["soil_temp"]);
    u8x8.drawString(1, 2, buffer);
    snprintf(buffer, sizeof(buffer), "Soil␣Moist:␣%.4f", node_doc["soil_moisture"]);
    u8x8.drawString(1, 3, buffer);
    snprintf(buffer, sizeof(buffer), "Soil␣EC:␣%.4f", node_doc["soil_ec"]);
    u8x8.drawString(1, 4, buffer);
    delay(20000);
    #endif

    //Acquire TEM from NiuBol
    #ifdef SOIL_NIUBOL_ENABLED
    acquire_soil_niubol(SOIL_NIUBOL_SLAVE_ID, site_id, soil_niubol_device_id, node_doc
    ["lat"], node_doc["lat"], node_doc["year"], node_doc["doy"], node_doc["hh"],
```

```
  node_doc["mm"], node_doc["ss"]);
  u8x8.clearDisplay();
  u8x8.setInverseFont(1);
  u8x8.drawString(1,0,"Soil␣TEM");
  u8x8.setInverseFont(0);
  snprintf(buffer, sizeof(buffer), "Soil␣Temp:␣%.4f", node_doc["soil_temp"]);
  u8x8.drawString(1, 2, buffer);
  snprintf(buffer, sizeof(buffer), "Soil␣Moist:␣%.4f", node_doc["soil_moisture"]);
  u8x8.drawString(1, 3, buffer);
  snprintf(buffer, sizeof(buffer), "Soil␣EC:␣%.4f", node_doc["soil_ec"]);
  u8x8.drawString(1, 4, buffer);
  delay(20000);
  #endif

  //Acquire Leaf data
  #ifdef LEAF_ENABLED
  acquire_leaf(LEAF_SLAVE_ID, site_id, leaf_device_id, node_doc["lat"], node_doc["
  lat"], node_doc["year"], node_doc["doy"], node_doc["hh"], node_doc["mm"], node_doc
  ["ss"]);
  u8x8.clearDisplay();
  u8x8.setInverseFont(1);
  u8x8.drawString(1,0,"Leaf");
  u8x8.setInverseFont(0);
  snprintf(buffer, sizeof(buffer), "Temp:␣%.4f", node_doc["leaf_temp"]);
  u8x8.drawString(1, 2, buffer);
  snprintf(buffer, sizeof(buffer), "Moist:␣%.4f", node_doc["leaf_wet"]);
  u8x8.drawString(1, 3, buffer);
  delay(20000);
  #endif


  // Serialize the JSON object
  // Add data to the JSON object

  filenameString = +"/" + String(site_id) + "_" + String(year) + "_" + String(doy) +
   ".json";

  String jsonString;
  char jsonBuffer[2048];
  serializeJson(node_doc, jsonString);

  // Print the JSON string
  Serial.println("JSON␣data:");
  Serial.println(jsonString);

  Serial.print("File␣Name:␣"); Serial.print(filenameString); Serial.print("\n");

  // Write the JSON data to a file
  if (!SD.exists(filenameString)) {
  Serial.println("File␣does␣not␣exist.␣Creating␣a␣new␣file.");
  createNewFile(filenameString.c_str());
}
  appendToFile(filenameString.c_str(), jsonString+"\n");

Serial.println("JSON␣string␣appended␣to␣file␣successfully!");

// Connect MQTT
if (!client.connected()) {
```

```
  reconnect();
}
client.loop();

//Publish MQTT
#ifdef PH_ENABLED
serializeJson(gateway_doc_ph, jsonBuffer);
client.publish(mqtt_topic_pH, jsonBuffer);
Serial.println("Published:␣");
Serial.println(jsonBuffer);
#endif

#ifdef NPK_ENABLED
serializeJson(gateway_doc_n, jsonBuffer);
client.publish(mqtt_topic_n, jsonBuffer);
Serial.println("Published:␣");
Serial.println(jsonBuffer);

serializeJson(gateway_doc_p, jsonBuffer);
client.publish(mqtt_topic_p, jsonBuffer);
Serial.println("Published:␣");
Serial.println(jsonBuffer);

serializeJson(gateway_doc_k, jsonBuffer);
client.publish(mqtt_topic_k, jsonBuffer);
Serial.println("Published:␣");
Serial.println(jsonBuffer);
#endif

#ifdef RAIN_ENABLED
serializeJson(gateway_doc_rain, jsonBuffer);
client.publish(mqtt_topic_rain, jsonBuffer);
Serial.println("Published:␣");
Serial.println(jsonBuffer);
#endif

#ifdef LIGHT_ENABLED
serializeJson(gateway_doc_light, jsonBuffer);
client.publish(mqtt_topic_light, jsonBuffer);
Serial.println("Published:␣");
Serial.println(jsonBuffer);
#endif

#ifdef SOIL_SEEED_ENABLED
serializeJson(gateway_doc_soil_temp_seeed, jsonBuffer);
client.publish(mqtt_topic_s_temp, jsonBuffer);
Serial.println("Published:␣");
Serial.println(jsonBuffer);

serializeJson(gateway_doc_soil_moist_seeed, jsonBuffer);
client.publish(mqtt_topic_s_moist, jsonBuffer);
Serial.println("Published:␣");
Serial.println(jsonBuffer);

serializeJson(gateway_doc_soil_ec_seeed, jsonBuffer);
client.publish(mqtt_topic_s_ec, jsonBuffer);
Serial.println("Published:␣");
Serial.println(jsonBuffer);
```

```
  #endif

  #ifdef SOIL_NIUBOL_ENABLED
  serializeJson(gateway_doc_soil_temp_niubol, jsonBuffer);
  client.publish(mqtt_topic_s_temp, jsonBuffer);
  Serial.println("Published:␣");
  Serial.println(jsonBuffer);

  serializeJson(gateway_doc_soil_moist_niubol, jsonBuffer);
  client.publish(mqtt_topic_s_moist, jsonBuffer);
  Serial.println("Published:␣");
  Serial.println(jsonBuffer);

  serializeJson(gateway_doc_soil_ec_niubol, jsonBuffer);
  client.publish(mqtt_topic_s_ec, jsonBuffer);
  Serial.println("Published:␣");
  Serial.println(jsonBuffer);
  #endif

  #ifdef LEAF_ENABLED
  serializeJson(gateway_doc_leaf_temp, jsonBuffer);
  client.publish(mqtt_topic_l_temp, jsonBuffer);
  Serial.println("Published:␣");
  Serial.println(jsonBuffer);

  serializeJson(gateway_doc_leaf_wet, jsonBuffer);
  client.publish(mqtt_topic_l_wet, jsonBuffer);
  Serial.println("Published:␣");
  Serial.println(jsonBuffer);
  #endif


}
```

4 | Gateway Development

Serving as a link between centralised systems and IoT nodes, a gateway is an essential part of the IoT ecosystem [41]. Gateways are used in smart agriculture to collect, process, and forward data from several nodes to local computers or cloud servers for additional analysis and decision-making. To effectively manage massive volumes of data, these devices are outfitted with strong processors, numerous connection interfaces, and reliable software. In order to optimise resource utilisation, increase crop yields, and lower operating costs, real-time monitoring and control of agricultural activities are made possible by gateways, which guarantee smooth communication and data transfer.

This chapter discusses the creation of Internet of Things (IoT) gateways specifically designed for smart agriculture. It offers a thorough examination of their hardware requirements, their place in the OSI model's Network and other subsequent layers, and how to write software for them using Python 3.12. It starts by examining the fundamental hardware components of an Internet of Things gateway, including CPUs, different peripherals and their interfacing. The chapter also looks at the vital roles that IoT gateways play in the Network, Transport and finally Session and Presentation layers, emphasising how important they are for supplying dependable connectivity and data transfer over agricultural landscapes. In addition, this chapter offers a thorough overview of the software development process for IoT gateways using Python 3.12. It covers the configuration of data processing methods, communication protocols, and cloud service integration for data analysis and storage.

4.1 | Hardware Features and Device Configuration

The IoT gateway in this smart agriculture application is built around the Raspberry Pi 4B [54], a powerful and versatile single-board computer. The key hardware components include:

- **Raspberry Pi 4B:** The Raspberry Pi 4B serves as the core of the gateway, equipped with a quad-core ARM Cortex-A72 processor, up to 8GB of RAM, and multiple connectivity options. Its features include:
  - **Processor and Memory:** A quad-core ARM Cortex-A72 processor running at 1.5GHz and up to 8GB of RAM, providing ample processing power for handling data processing and communication tasks.
  - **USB Ports:** Four USB ports (two USB 3.0 and two USB 2.0) allow connection of various peripherals, including storage devices and communication modules.
  - **HDMI Output:** Dual micro-HDMI ports support up to 4K resolution, allowing the connection of high-definition displays like the Waveshare 5.5-inch LCD.
  - **Ethernet and WiFi Connectivity:** Gigabit Ethernet port for wired network connections and dual-band 802.11ac WiFi for wireless connectivity.
  - **GPIO Pins:** 40-pin GPIO header for connecting various sensors and actuators, enabling customization and expansion of the gateway's functionality.
- **Waveshare 5.5-inch LCD Display [1]:** The Waveshare 5.5-inch LCD display provides a user-friendly interface for monitoring and controlling the gateway. Its features include:
  - **Resolution:** 1920x1080 full HD resolution, ensuring clear and detailed visualization of data.
  - **Touchscreen:** Capacitive touchscreen functionality allows for intuitive interaction with the gateway's user interface.
  - **Connection:** The display connects to the Raspberry Pi via an HDMI interface, with a USB connection for touch control.
- **SIM7600X G-H Hat [4]:** The SIM7600X G-H hat is a cellular communication module that enhances the connectivity of the gateway. Its features include:
  - **4G LTE Connectivity:** Provides high-speed internet access, enabling the gateway to communicate with cloud services and other remote systems even in areas without WiFi or Ethernet.
  - **GNSS (Global Navigation Satellite System):** Supports GPS, GLONASS, and BeiDou, providing accurate geolocation data for precise positioning and timing information.

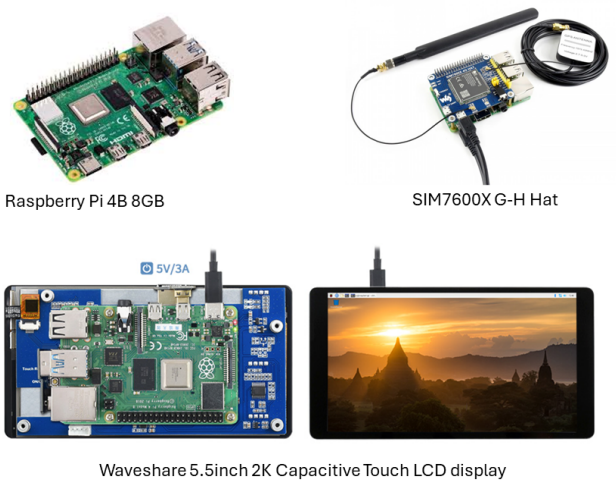

Figure 4.1: Components used to build the gateway.

- **UART and USB Interface:** Connects to the Raspberry Pi via UART or USB, ensuring flexible and reliable data transmission.

- **Primary Power Source:** The Raspberry Pi can be powered via its USB-C power port, typically using a 5V/3A power adapter. This powers up all the peripherals attached to the Raspberry Pi.

Figure 4.1 shows the pictures of all the components used in ideating the IoT gateway. These components are then assembled in a custom enclosure designed by the CSIO Mechanical Design and Fabrication facility. The drawings of these designs can be refered from Appendix C.
Before getting into the development of the gateway software, the device components in the gateway need to be configured appropriately. Initially, the Raspberry Pi has to be loaded with the Raspbian OS as per the procedure outlined in [3] with a micro-HDMI monitor, a keyboard, and a mouse as peripherals. Once the Raspberry Pi can be used as a desktop, using the **raspi-config** command in the terminal, all the interfaces for UART, SPI, I2C, and GPIO are to be enabled, along with connection services using SSH. The procedure to configure them can be found in [3]. After this, the Raspberry Pi needs to be configured as a WiFi hotspot, so that it can work as an Access Point (AP). For the purpose of this HCP project, the AP is named as **RSSA_AP**, with the password as **RSSA_12345678**. The Raspberry Pi is configured as a WiFi AP using a Python script that configures and enables the **hostapd, dnsmasq** and **dhcpcd** services, and disables the Network Manager services in the Raspberry Pi for the AP to operate. The following code block does this:

```
import subprocess

def run_command(command):
    result = subprocess.run(command, shell=True, text=True, capture_output=True)
    if result.returncode != 0:
        print(f"Error running command: {command}\n{result.stderr}")
        raise Exception(f"Command failed: {command}")

def setup_hostapd(ssid, password):
    hostapd_conf = f"""
interface=wlan0
driver=nl80211
ssid={ssid}
hw_mode=g
channel=6
wmm_enabled=0
macaddr_acl=0
auth_algs=1
```

```
ignore_broadcast_ssid=0
wpa=2
wpa_passphrase={password}
wpa_key_mgmt=WPA-PSK
rsn_pairwise=CCMP
"""
    with open("/etc/hostapd/hostapd.conf", "w") as file:
        file.write(hostapd_conf)
    with open("/etc/default/hostapd", "w") as file:
        file.write('DAEMON_CONF="/etc/hostapd/hostapd.conf"')

def setup_dnsmasq():
    dnsmasq_conf = """
interface=wlan0
dhcp-range=192.168.4.2,192.168.4.20,255.255.255.0,24h
    """
    with open("/etc/dnsmasq.conf", "w") as file:
        file.write(dnsmasq_conf)

def configure_network():
    dhcpcd_conf = """
interface wlan0
    static ip_address=192.168.4.1/24
    nohook wpa_supplicant
    """
    with open("/etc/dhcpcd.conf", "a") as file:
        file.write(dhcpcd_conf)

def start_services():
    run_command("sudo systemctl restart dhcpcd")
    run_command("sudo systemctl unmask hostapd")
    run_command("sudo systemctl enable hostapd")
    run_command("sudo systemctl start hostapd")
    run_command("sudo systemctl enable dnsmasq")
    run_command("sudo systemctl start dnsmasq")

def create_wifi_ap(ssid, password):
    try:
        setup_hostapd(ssid, password)
        setup_dnsmasq()
        configure_network()
        start_services()
        print("Wi-Fi SoftAP created successfully.")
    except Exception as e:
        print(f"Failed to create Wi-Fi SoftAP: {e}")

if __name__ == "__main__":
    ssid = "RSSA_AP"
    password = "RSSA_12345678"
    create_wifi_ap(ssid, password)
```

4.2 | Data Reception from the Nodes

In an IoT architecture where a Raspberry Pi 4B functions as a gateway, data reception from various sensors or nodes is a critical task. For this purpose, MQTT (Message Queuing Telemetry Transport) [26] is a commonly used protocol due to its lightweight nature and efficiency in handling real-time data streams. MQTT is a publish/subscribe messaging protocol designed for lightweight communication between devices in a network. It operates on a client-server architecture where the server, known as the MQTT broker,

manages the distribution of messages. Clients, such as the ESP32 nodes and the Raspberry Pi gateway, connect to the broker to either publish or subscribe to data topics.

The MQTT broker acts as the central hub for message exchange. On the Raspberry Pi, a popular choice for an MQTT broker is Mosquitto [15]. This broker facilitates communication by receiving messages from publishing clients and delivering them to subscribing clients based on the topics of interest.

- **Broker Setup:** To establish the MQTT broker on the Raspberry Pi, Mosquitto is installed and configured. By default, Mosquitto listens for incoming connections on port 1883, which is used for MQTT communication.
- **Broker Operation:** Once set up, the Mosquitto service handles incoming messages from MQTT clients, such as the ESP32 nodes. It routes these messages to clients that are subscribed to the corresponding topics.

The ESP32 node, equipped with Wi-Fi capabilities, acts as an MQTT client that publishes data to the broker. It collects sensor data and transmits it to the MQTT broker using specified topics. The ESP32 node's role is to publish data to a topic that the Raspberry Pi gateway subscribes to. The Raspberry Pi is subscribed to all the topics listed in Table 3.1.

- **Data Publishing:** The ESP32 is programmed to connect to the MQTT broker and publish data at regular intervals. This data is associated with a topic that is meaningful for the application, such as "pH_data" or "s_moist_data."
- **Connectivity:** The ESP32 establishes a connection with the MQTT broker over the network, sending data packets that include sensor readings or other relevant information.

On the Raspberry Pi, an MQTT client subscribes to the topics of interest to receive data published by the ESP32 nodes. This subscription allows the gateway to process and act on the incoming data.

- **Subscription:** The MQTT client on the Raspberry Pi subscribes to one or more topics that correspond to the data published by the ESP32 nodes. This subscription ensures that any message sent to these topics is delivered to the Raspberry Pi.
- **Data Handling:** Once the Raspberry Pi receives the messages, it can process the data as required. This might include logging the data, performing analysis, or integrating it with other systems.
- **Real-time Updates:** MQTT's efficient message delivery ensures that data from the ESP32 nodes is received in near real-time, facilitating timely responses and actions based on the sensor readings.

4.3 | Data Processing in the Gateway

Once the MQTT client on the Raspberry Pi receives data in the form of JSON strings, it is crucial to process and store this data efficiently for future analysis and use. This section outlines the approach to parse the incoming JSON data and organize it into local storage based on the node device and the day of the year.

The first step in data processing is parsing the JSON strings received from the MQTT broker. JSON is a widely-used data format that is both human-readable and machine-friendly. Each JSON string typically contains key-value pairs that represent the sensor readings and metadata from the ESP32 nodes.

A received string would look something like this:

```
{
  "site_id": "RSSA_BLR_01",
  "device_id": "NiuBol_DKI_NPK_01",
  "lat": 30.71,
  "lon": 76.79,
  "year": 2024,
  "doy": 178,
  "time": "03:22:22",
  "Quantity": "Phosphorous",
  "Value": 616.8,
  "Unit": "mg/kg"
}
```

This JSON object includes information about the node (**site_id**), the year and day of year (**year** and **doy** respectively). Apart from this, the observable and its value is included in the JSON string. The Raspberry Pi segregates the received JSON objects based initially on the **site_id** of the JSON object, then on the basis of the field **year**, then followed by **doy**. Therefore, the file path of the storage of the JSON objects becomes **{parent_directory}/{site_id}/{year}/{doy}.json**. Therefore, based on the aforementioned fields, each incoming JSON string is appended to their respective JSON files. Such kind of an organization leads to efficient data management, as by organizing data based on the node device and the day of the year, retrieval and analysis become straightforward and efficient. Also, historical data for each node is easily accessible and well-organized, facilitating trend analysis and reporting. This method supports scalability as the number of nodes and the volume of data increase. Each node's data is compartmentalized, reducing complexity. With data stored in JSON format and organized by day, it is easy to load and process specific data sets as required for various applications.

After the organization of data, there is a copy of the incoming gateway stored in the scope of the gateway software so that it can be used for data transmission to the cloud.

4.4 | Transmission to the Cloud

The Raspberry Pi, a versatile single-board computer, can be paired with the SIM7600X G-H Hat to enable cellular connectivity, making it possible to send data to the cloud from virtually anywhere. This section will explore various methods for sending JSON data to the cloud, starting with an overview of using the SIM7600X G-H Hat, followed by instructions for using HTTPS, and concluding with a detailed example of sending data to the ThingSpeak cloud using HTTPS POST.

The SIM7600X G-H Hat is a 4G/3G/2G communication hat designed for the Raspberry Pi, providing high-speed cellular connectivity. It supports multiple modes of communication, including SMS, voice, and data transfer. Here are the key steps to get started with sending data using this hat:

1. **Hardware Setup:**
   - Attach the SIM7600X G-H Hat to the Raspberry Pi's GPIO pins.
   - Insert a SIM card with an active data plan into the hat.
   - Connect the antenna to ensure good signal reception.
2. **Software Setup:**
   - Install necessary drivers and software packages, such as PPP, screen, and minicom, to manage the cellular connection.
   - Configure PPP (Point-to-Point Protocol) to establish the data connection using the cellular network. This involves setting up configuration files that specify the APN (Access Point Name) provided by your mobile carrier.
3. **Establishing the Cellular Connection:** Use appropriate commands to initiate the PPP connection, which will establish the cellular link and provide internet access to the Raspberry Pi. Verify the connection by checking the network interface status.

HTTPS (HyperText Transfer Protocol Secure) is an extension of HTTP, ensuring secure communication over a computer network. Using HTTPS to send data adds a layer of encryption, protecting the data from interception and tampering during transmission. Here's how to set up HTTPS communication on the Raspberry Pi:

1. **Install Required Tools:** Tools like cURL can be used to send HTTP and HTTPS requests. Install cURL on the Raspberry Pi to facilitate data transmission.
2. **Sending Data Using HTTPS POST:** Use cURL to send JSON data securely to a server via an HTTPS POST request. This involves specifying the URL of the server, setting the appropriate headers to indicate the content type, and including the JSON data in the request body.

ThingSpeak is a popular IoT analytics platform that allows users to aggregate, visualize, and analyze live data streams. It provides an easy way to send and store data from various IoT devices, making it ideal for cloud-based data logging and analysis. Follow these steps to send JSON data to ThingSpeak:

1. **Create a ThingSpeak Account:** Sign up for a free account on ThingSpeak and create a new channel. Each channel can hold multiple fields of data, and ThingSpeak provides API keys for secure data submission.
2. **Obtain Write API Key:** After creating a channel, obtain the Write API Key from the channel settings. This key is used to authenticate your data submissions and ensures that only authorized data is accepted.
3. **Send Data Using HTTPS POST:** Use a tool like cURL to send data to ThingSpeak via an HTTPS POST request. This involves specifying the ThingSpeak API endpoint, including the Write API Key for authentication, and sending the data fields as part of the request.

By following these steps, you can effectively send JSON data from your Raspberry Pi to the cloud using the SIM7600X G-H Hat and HTTPS, with a practical example of integrating with the ThingSpeak platform. This process ensures secure and reliable data transmission, enabling various IoT applications that require remote data monitoring and analysis. Here is a pseudocode translated into Python code using the **requests** library when the Raspberry Pi is already connected to the internet:

```
import requests

# Define the ThingSpeak API endpoint
thingspeak_url = "https://api.thingspeak.com/update"

# Define the Write API Key obtained from ThingSpeak
write_api_key = "YOUR_API_KEY"

# Define the data to be sent
field1_value = "value1"
field2_value = "value2"

# Create the data payload
data_payload = {
    'api_key': write_api_key,
    'field1': field1_value,
    'field2': field2_value
}

# Send the HTTP POST request
response = requests.post(thingspeak_url, data=data_payload)

# Check the response status code
if response.status_code == 200:
    print("Data successfully sent to ThingSpeak")
else:
    print(f"Failed to send data. HTTP status code: {response.status_code}")
```

4.5 | Software Development of the Gateway

The software for the gateway hardware in smart agriculture is developed using Python. Python is chosen due to its simplicity, extensive libraries, and robust community support, making it an ideal language for rapid development and deployment in IoT applications. In the context of smart agriculture, the gateway hardware operates as a real-time embedded system. This means it must process data and respond to events within a specified time constraint to ensure the proper functioning of the entire system. Some key principles of real-time embedded systems [48] applied in our software development include:

- **Determinism:** The system must perform operations and produce results in a predictable manner. This is crucial for tasks such as sensor data acquisition and control signal generation.
- **Concurrency:** The system handles multiple tasks simultaneously, such as monitoring various sensors, processing data, and communicating with the cloud. This is achieved through efficient task scheduling and management.

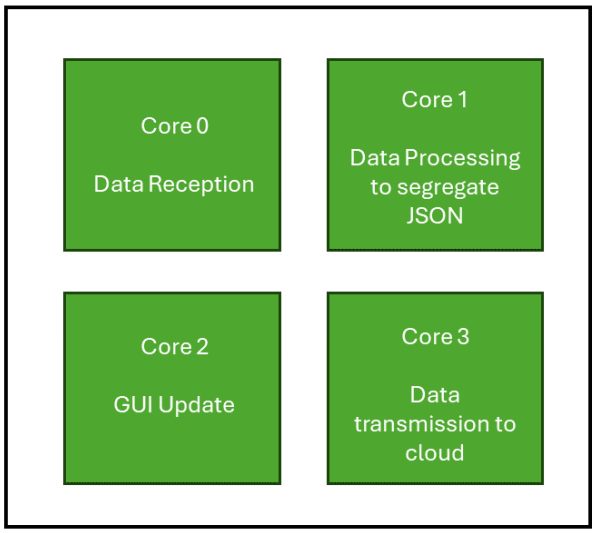

Figure 4.2: Assignment of thread functions to different cores in the Raspberry Pi gateway.

- **Reliability and Fault Tolerance:** The software is designed to handle faults and continue operating under adverse conditions. This includes implementing error-handling routines and ensuring data integrity during transmission and storage.
- **Efficiency:** The software must be optimized to use the limited resources of the embedded hardware effectively. This involves efficient memory management, minimizing power consumption, and optimizing processing time.

In this section, the software development for the gateway starts out by first designing the threads that will be executed to perform the designated functions of the gateway, that are data reception, data processing, GUI update, and data transmission. The subsequent subs-sections will be discussing these threads, and finally how they are put together to create a functioning gateway.

4.5.1 | Multithreading and Core Allocation

Multithreading is a programming technique that allows multiple threads to run concurrently within a single process. In the context of gateway hardware for smart agriculture, multithreading enables the system to perform various tasks simultaneously, such as reading sensor data, processing information, and communicating with the cloud. By using Python's threading module, the software can create and manage multiple threads, ensuring that the gateway remains responsive and can handle multiple operations without significant delays.

Modern gateway hardware often features multi-core processors, allowing the software to achieve true parallelism by distributing threads across different cores [48]. This means multiple threads can execute simultaneously on separate cores, significantly improving the system's overall performance and throughput. Allocating specific cores to handle distinct tasks, such as dedicating one core to sensor data acquisition, another to data processing, and a third to communication tasks, prevents any single core from becoming a bottleneck. This efficient allocation ensures higher system efficiency, better resource utilization, and improved responsiveness, which are essential for the demanding tasks in a smart agriculture environment.

To implement parallelism of processes performing data reception, data processing, GUI update and data transmission, four threads are implemented, each to perform the aforementioned functions. The Raspberry Pi has 4 cores. For all four threads to run concurrently, the thread for data reception is assigned to core-1 (indexed 0), the thread for data processing to core-2 (indexed 1), the thread for GUI update to core-3 (indexed 2), and the thread for data transmission to core-4 (indexed 3). This is shown by the diagram in Figure 4.2. The following code block is an example of how to create new threads, and assign them to different cores of the Raspberry Pi.

```
### Do not execute as is
### Define the required functions first
```

```
import threading
import os
import time
import psutil
from datetime import datetime
import time
import subprocess
import signal
import sys

# Function to set thread affinity
def set_thread_affinity(thread, core_id):
    p = psutil.Process(thread.ident)
    p.cpu_affinity([core_id])

# Main function
if __name__ == "__main__":
    try:

        mqtt_thread = threading.Thread(target=start_mqtt_broker)
        set_thread_affinity(mqtt_thread, 0)
        processing_thread = threading.Thread(target=json_processing_thread)
        set_thread_affinity(processing_thread, 1)
        gui_update_thread = threading.Thread(target=update_gui, args=(
    data_value_placeholders, data_unit_placeholders, node_var))
        set_thread_affinity(gui_update_thread, 2)
        send_http_thread = threading.Thread(target=update_cloud)
        set_thread_affinity(send_http_thread, 3)

        mqtt_thread.start()
        processing_thread.start()
        gui_update_thread.start()
        send_http_thread.start()

        root.mainloop()

        mqtt_thread.join()
        processing_thread.join()
        gui_update_thread.join()
        send_http_thread.join()

    except Exception as e:
        print(f"An␣error␣occurred:␣{e}")
        send_at('AT+CIPCLOSE=0', '+CIPCLOSE:␣0,0', 15)
        send_at('AT+NETCLOSE', '+NETCLOSE:␣0', 1)
        GPIO.cleanup()
        ser.close()
        sys.exit(1)
```

4.5.2 | Data Reception

This subsection explains the functions that your Python script has to define for accepting and processing MQTT messages on a Raspberry Pi. It is made sure that the script is modular, legible, and manageable by breaking the code up into clearly defined functions.

1. **MQTT Client Setup:** The first step is to set up the MQTT client, which involves defining functions to handle the connection, subscriptions, and callbacks for incoming messages.

- **connect_mqtt():** This function establishes a connection to the MQTT broker. This function creates an instance of the MQTT client, defines the connection parameters (broker address, port, keep-alive interval) and handles connection success or failure.
- **on_connect(client, userdata, flags, rc):** It is a callback function that is triggered when the client connects to the MQTT broker. It checks the connection result code **(rc)**. It also logs connection success or handle reconnection attempts if the connection fails.

2. **Subscription Handling:** Once connected, the client needs to subscribe to the relevant topics to receive data.
   - **subscribe_topics(client):** It subscribes to one or more MQTT topics. The function defines the topics and their Quality of Service (QoS) levels and also ensures that subscriptions are confirmed.
   - **on_subscribe(client, userdata, mid, granted_qos):** It is a callback function that confirms the subscription to the topics. It logs the successful subscription and its QoS level.
3. **Message Processing:** Handling the incoming messages requires a function to process and store the data appropriately.
   - **on_message(client, userdata, msg):** It is a callback function that processes incoming messages. It parses the received message, identifies the source node and extract relevant data.

Not all of the aforementioned functions are necessary to establish an MQTT connection and receive data using it. The following code block shows in Python how to initialize an MQTT broker using Mosquitto, and initialize a client with some of the necessary callback functions to start receiving data.

```
### Do not execute the code as is

import paho.mqtt.client as mqtt
import json
from collections import deque
import threading
import os
import json
import time
import psutil
from datetime import datetime
import serial
import RPi.GPIO as GPIO
import time
import os
import subprocess
import requests
import signal
import sys

MQTT_ADDRESS = "192.168.4.1"
MQTT_USER = "rssa-gateway"
MQTT_PASSWORD = "rssa_gateway"

class CircularQueue:
    def __init__(self, size):
        self.size = size
        self.buffer = [None] * size
        self.head = 0
        self.tail = 0
        self.count = 0
        self.lock = threading.Lock()

    def put(self, item):
```

```
        with self.lock:
            self.buffer[self.tail] = item
            self.tail = (self.tail + 1) % self.size
            if self.count == self.size:
                self.head = (self.head + 1) % self.size  # Overwrite the oldest data
            else:
                self.count += 1

    def get(self):
        with self.lock:
            if self.count == 0:
                return None
            item = self.buffer[self.head]
            self.head = (self.head + 1) % self.size
            self.count -= 1
            return item

    def is_empty(self):
        with self.lock:
            return self.count == 0

RING_BUFFER_SIZE = 200
ring_buffer_in = CircularQueue(RING_BUFFER_SIZE)

# Function to set thread affinity
def set_thread_affinity(thread, core_id):
    p = psutil.Process(thread.ident)
    p.cpu_affinity([core_id])

def on_connect(client, userdata, flags, rc):
    print("Connected␣with␣Result␣Code", str(rc))
    client.subscribe("pH_data")
    client.subscribe("n_data")
    client.subscribe("p_data")
    client.subscribe("k_data")
    client.subscribe("rain_data")
    client.subscribe("light_data")
    client.subscribe("s_temp_data")
    client.subscribe("s_moist_data")
    client.subscribe("s_ec_data")
    client.subscribe("l_temp_data")
    client.subscribe("l_wet_data")

def on_message(client, userdata, msg):
#    print("Topic:", msg.topic)
#    print("Message:", msg.payload.decode())
    payload = msg.payload.decode('utf-8')
    data = json.loads(msg.payload)
    ring_buffer_in.put(data)
    with open("gateway_data_rx.json", "a") as f:
        f.write(json.dumps(data) + "\n")

def start_mqtt_broker():
    client = mqtt.Client()
    client.username_pw_set(MQTT_USER, MQTT_PASSWORD)
    client.on_connect = on_connect
    client.on_message = on_message
    client.connect("192.168.4.1", 1883)
```

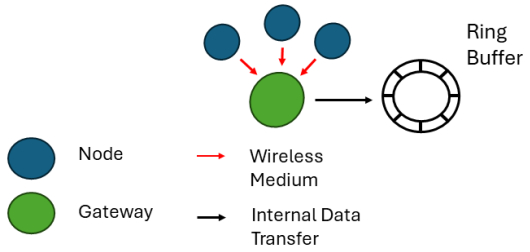

Figure 4.3: Flow of operations for data reception.

```
    client.loop_forever()

# Main function
if __name__ == "__main__":
    try:
        mqtt_thread = threading.Thread(target=start_mqtt_broker)
        set_thread_affinity(mqtt_thread, 0)
        mqtt_thread.start()
        mqtt_thread.join()

    except Exception as e:
        print(f"An␣error␣occurred:␣{e}")
        sys.exit(1)
```

After receiving the data using MQTT, the same thread pushes the data into a ring buffer[29]. Ring buffers are used extensively in the implementation of the gateway logic so that there is no intermediary loss of data when it is being transferred from the domain of one thread to the next. The next thread of data processing accesses data from this ring buffer defined as **ring_bufffer_in**. The overall flow of operations in this thread is shown in Figure 4.3.

4.5.3 | Data Processing

After the incoming data is pushed into the ring buffer **ring_bufffer_in**, a second thread to process the incoming data runs concurrently. This thread segregates the incoming data based on the value of the fields **site_id**, and then the JSON object is appended to the file **{parent_directory}/{site_id}/{year}/{doy}.json**. In this way, the incoming data is divided into several structured sources that can be later used to form a database.Figure 4.4 shows the function of this thread. The following code block provides a snippet with functions to process the incoming data. After processing the JSON, they are transferred to **ring_bufffer_gui** so that it can be used by the subsequent thread.

```
### Do not execute the code as is!

import paho.mqtt.client as mqtt
import json
from collections import deque
import threading
import os
import json
import pandas as pd
import tkinter as tk
from tkinter import ttk
from tkinter import font as tkfont
import time
import psutil
from PIL import Image, ImageTk
from datetime import datetime
import serial
```

```
import RPi.GPIO as GPIO
import time
import os
import subprocess
import requests
import signal
import sys

class CircularQueue:
    def __init__(self, size):
        self.size = size
        self.buffer = [None] * size
        self.head = 0
        self.tail = 0
        self.count = 0
        self.lock = threading.Lock()

    def put(self, item):
        with self.lock:
            self.buffer[self.tail] = item
            self.tail = (self.tail + 1) % self.size
            if self.count == self.size:
                self.head = (self.head + 1) % self.size  # Overwrite the oldest data
            else:
                self.count += 1

    def get(self):
        with self.lock:
            if self.count == 0:
                return None
            item = self.buffer[self.head]
            self.head = (self.head + 1) % self.size
            self.count -= 1
            return item

    def is_empty(self):
        with self.lock:
            return self.count == 0

RING_BUFFER_SIZE = 200
ring_buffer_in = CircularQueue(RING_BUFFER_SIZE)
ring_buffer_gui = CircularQueue(RING_BUFFER_SIZE)

# Function to set thread affinity
def set_thread_affinity(thread, core_id):
    p = psutil.Process(thread.ident)
    p.cpu_affinity([core_id])

def json_processing_thread():
    while True:
        json_obj = ring_buffer_in.get()
        if json_obj is not None:
            process_json(json_obj)
            #send_to_thingspeak(json_obj)
            ring_buffer_gui.put(json_obj)

def process_json(json_obj):
    site = json_obj['site_id']
```

Figure 4.4: Data processing thread flow.

```
    year = json_obj['year']
    doy = json_obj['doy']

    # Create directory path
    dir_path = os.path.expanduser(f"~/RSSA_data_log/{site}/{year}")
    os.makedirs(dir_path, exist_ok=True)

    # File path
    file_path = os.path.join(dir_path, f"{doy}.json")

    # Append JSON object to file
    with open(file_path, 'a') as f:
        f.write(json.dumps(json_obj) + "\n")

    #print(f"Processed and saved JSON: {json_obj}")

# Main function
if __name__ == "__main__":
    try:
        processing_thread = threading.Thread(target=json_processing_thread)
        set_thread_affinity(processing_thread, 1)
        processing_thread.start()
        processing_thread.join()

    except Exception as e:
        print(f"An␣error␣occurred:␣{e}")
        sys.exit(1)
```

4.5.4 | GUI Development

For standalone operation by a user, the gateway device comes with an LCD display with capacitive touch to display data. A GUI is developed using the **tkinter** library of Python to initialize and update the GUI with incoming data. The GUI has a few major components:

- Logos of CSIR and CSIO.
- Name of the GUI to be displayed on the top (KRISHI-IoT Gateway, Designed and Developed by CSIR-CSIO Chandigarh).
- Current Date and Time.
- Dropdown list for available site IDs.
- An area to display Latitude, Longitude, time at which the data is acquired by the node, day of year and current year.
- An area to display Soil pH, Soil NPK.

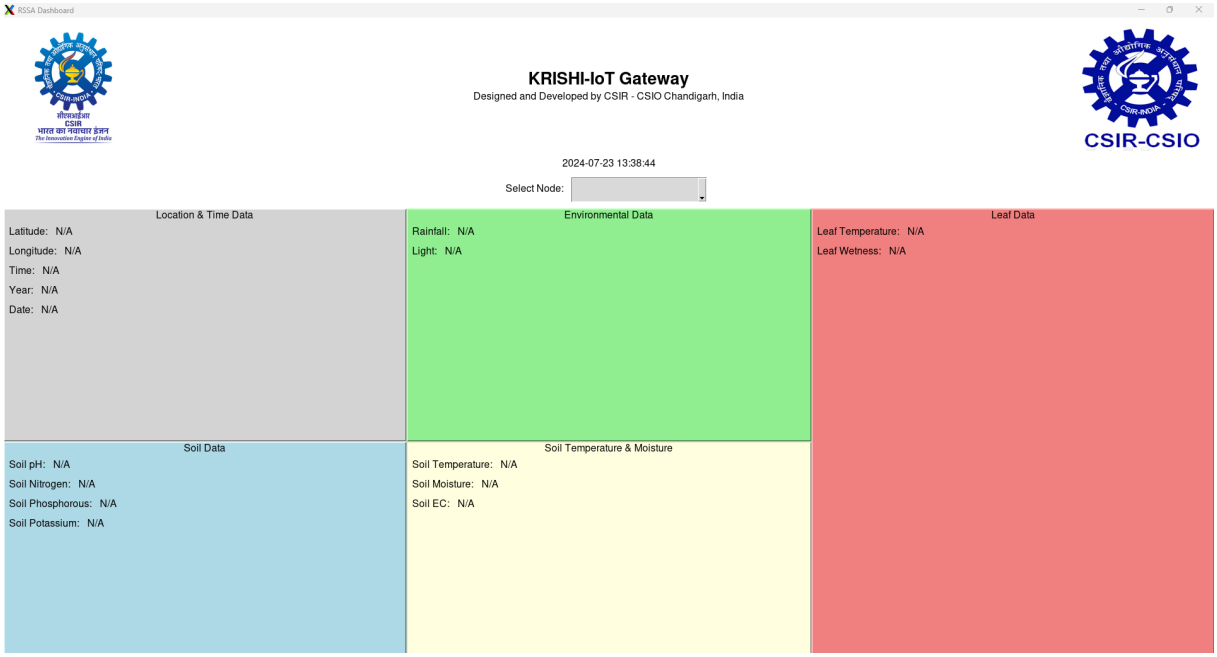

Figure 4.5: GUI after initialization.

- An area to display soil Moisture, soil EC, soil temperature.
- An area to display meteorological data.
- An area to display leaf temperature and leaf wetness.

At first, a **start_gui()** function is called to initialize the GUI. It is such that it puts the CSIR logo on the top left, and CSIO logo on the top right. In between the logo, the phrase **KRISHI-IoT Gateway** is written. Below this, the phrase **Designed and Developed by CSIR-CSIO, Chandigarh** is mentioned. Below this, the current date and time is displayed. After this, a dropdown box is inserted that provides the list of available nodes from which the data can be received. Below this, all the 5 areas to show the measurements from the nodes are initialized in a grid fashion. The initialized GUI will look like as shown in Figure 4.5.
Now, a thread to continuously update the GUI is started. This thread is attached to the **update_gui()** function. This thread takes data from the ring buffer **ring_bufffer_gui** and updates the current time and the incoming data with their units to be displayed in the 5 areas initialized based on the site ID selected from the dropdown list. After an update, the GUI can look something like shown in Figure 4.6. After the GUI is updated, the data is pushed to the ring buffer **ring_bufffer_out**. This is done so that the next thread to transmit data can fetch them without loss. The following code blocks defines these functions in Python.

```
### Do not execute the code block as is

def update_time(label):
    def update():
        current_time = datetime.now().strftime("%Y-%m-%d␣%H:%M:%S")
        label.config(text=current_time)
        label.after(1000, update)
    update()

def start_gui():
    root = tk.Tk()
    root.title("RSSA␣Dashboard")

    # Maximize window
    root.attributes('-zoomed', True)
```

```
    root.configure(bg='white')  # Set background color to white

    # Top Frame for logos and text
    top_frame = tk.Frame(root, bg='white')
    top_frame.pack(fill=tk.X)

    left_logo = Image.open("CSIR_logo.png")  # Update with actual path
    left_logo = left_logo.resize((200, 200), Image.ANTIALIAS)
    left_logo = ImageTk.PhotoImage(left_logo)
    left_logo_label = tk.Label(top_frame, image=left_logo, bg='white')
    left_logo_label.image = left_logo
    left_logo_label.pack(side=tk.LEFT, padx=10, pady=10)

    right_logo = Image.open("CSIO_logo.png")  # Update with actual path
    right_logo = right_logo.resize((200, 200), Image.ANTIALIAS)
    right_logo = ImageTk.PhotoImage(right_logo)
    right_logo_label = tk.Label(top_frame, image=right_logo, bg='white')
    right_logo_label.image = right_logo
    right_logo_label.pack(side=tk.RIGHT, padx=10, pady=10)

    middle_frame = tk.Frame(top_frame, bg='white')
    middle_frame.pack(expand=True)

    title_font = tkfont.Font(family="Helvetica", size=20, weight="bold")
    subtitle_font = tkfont.Font(family="Helvetica", size=12)

    title_label = tk.Label(middle_frame, text="KRISHI-IoT␣Gateway", font=title_font,
    bg='white')
    title_label.pack()

    subtitle_label = tk.Label(middle_frame, text="Designed␣and␣Developed␣by␣CSIR␣-␣
    CSIO␣Chandigarh,␣India", font=subtitle_font, bg='white')
    subtitle_label.pack()

    style = ttk.Style()

    # Configure the style to add padding and change the font size
    style.configure("TCombobox", padding=10, font=('Helvetica', 12))  # Adjust padding
     and font as needed

    # Date and time
    datetime_label = tk.Label(root, font=subtitle_font, bg='white')
    datetime_label.pack()
    update_time(datetime_label)

    # Node selection
    node_frame = tk.Frame(root, bg='white')
    node_frame.pack(pady=10)

    node_label = tk.Label(node_frame, text="Select␣Node:", font=subtitle_font, bg='
    white')
    node_label.pack(side=tk.LEFT)

    node_var = tk.StringVar()
    root.option_add("*TCombobox*Listbox*Font", subtitle_font)
    node_dropdown = ttk.Combobox(node_frame, textvariable=node_var, state="readonly",
    style="TCombobox", font = subtitle_font)
    node_dropdown['values'] = ["RSSA_CHD_01", "RSSA_BLR_01"]  # Example values, update
```

```
 with actual site IDs
node_dropdown.pack(side=tk.LEFT, padx=10)

data_value_placeholders = {}; data_unit_placeholders = {}

# Main frame for the data sections
main_frame = tk.Frame(root, bg='white')
main_frame.pack(fill=tk.BOTH, expand=True)

# Create frames for five sections
lat_lon_frame = tk.Frame(main_frame, bg='lightgray', relief=tk.RAISED, bd=2)
soil_frame = tk.Frame(main_frame, bg='lightblue', relief=tk.RAISED, bd=2)
env_frame = tk.Frame(main_frame, bg='lightgreen', relief=tk.RAISED, bd=2)
soil_temp_frame = tk.Frame(main_frame, bg='lightyellow', relief=tk.RAISED, bd=2)
leaf_frame = tk.Frame(main_frame, bg='lightcoral', relief=tk.RAISED, bd=2)

# Use grid to place frames in a 3x2 pattern
lat_lon_frame.grid(row=0, column=0, sticky="nsew")
soil_frame.grid(row=1, column=0, sticky="nsew")
env_frame.grid(row=0, column=1, sticky="nsew")
soil_temp_frame.grid(row=1, column=1, sticky="nsew")
leaf_frame.grid(row=0, column=2, rowspan=2, sticky="nsew")

main_frame.grid_rowconfigure(0, weight=1)
main_frame.grid_rowconfigure(1, weight=1)
main_frame.grid_columnconfigure(0, weight=1)
main_frame.grid_columnconfigure(1, weight=1)
main_frame.grid_columnconfigure(2, weight=1)

# Add content to lat_lon_frame
lat_lon_title = tk.Label(lat_lon_frame, text="Location␣&␣Time␣Data", font=
subtitle_font, bg='lightgray')
lat_lon_title.pack(anchor='n')

lat_lon_labels = [("Latitude", 0), ("Longitude", 1), ("Time", 2), ("Year", 3), ("
Date", 4)]
for label_text, _ in lat_lon_labels:
    frame = tk.Frame(lat_lon_frame, bg='lightgray')
    frame.pack(fill=tk.X, padx=5, pady=5)

    label = tk.Label(frame, text=label_text+":␣", font=subtitle_font, bg='
lightgray')
    label.pack(side=tk.LEFT)

    value = tk.Label(frame, text="N/A", font=subtitle_font, bg='lightgray')  #
Placeholder for actual values
    value.pack(side=tk.LEFT, padx=5)

    unit = tk.Label(frame, text="", font=subtitle_font, bg='lightgray')  #
Placeholder for actual units
    unit.pack(side=tk.LEFT, padx=5)

    data_value_placeholders[label_text] = value
    data_unit_placeholders[label_text] = unit

# Add content to soil_frame
soil_title = tk.Label(soil_frame, text="Soil␣Data", font=subtitle_font, bg='
lightblue')
```

```
    soil_title.pack(anchor='n')

    soil_labels = [("Soil␣pH", 0), ("Soil␣Nitrogen", 1), ("Soil␣Phosphorous", 2), ("
    Soil␣Potassium", 3)]
    for label_text, _ in soil_labels:
        frame = tk.Frame(soil_frame, bg='lightblue')
        frame.pack(fill=tk.X, padx=5, pady=5)

        label = tk.Label(frame, text=label_text+":␣", font=subtitle_font, bg='
    lightblue')
        label.pack(side=tk.LEFT)

        value = tk.Label(frame, text="N/A", font=subtitle_font, bg='lightblue')  #
    Placeholder for actual values
        value.pack(side=tk.LEFT, padx=5)

        unit = tk.Label(frame, text="", font=subtitle_font, bg='lightblue')  #
    Placeholder for actual units
        unit.pack(side=tk.LEFT, padx=5)

        data_value_placeholders[label_text] = value
        data_unit_placeholders[label_text] = unit

    # Add content to env_frame
    env_title = tk.Label(env_frame, text="Environmental␣Data", font=subtitle_font, bg='
    lightgreen')
    env_title.pack(anchor='n')

    env_labels = [("Rainfall", 0), ("Light", 1)]
    for label_text, _ in env_labels:
        frame = tk.Frame(env_frame, bg='lightgreen')
        frame.pack(fill=tk.X, padx=5, pady=5)

        label = tk.Label(frame, text=label_text+":␣", font=subtitle_font, bg='
    lightgreen')
        label.pack(side=tk.LEFT)

        value = tk.Label(frame, text="N/A", font=subtitle_font, bg='lightgreen')  #
    Placeholder for actual values
        value.pack(side=tk.LEFT, padx=5)

        unit = tk.Label(frame, text="", font=subtitle_font, bg='lightgreen')  #
    Placeholder for actual units
        unit.pack(side=tk.LEFT, padx=5)

        data_value_placeholders[label_text] = value
        data_unit_placeholders[label_text] = unit

    # Add content to soil_temp_frame
    soil_temp_title = tk.Label(soil_temp_frame, text="Soil␣Temperature␣&␣Moisture",
    font=subtitle_font, bg='lightyellow')
    soil_temp_title.pack(anchor='n')

    soil_temp_labels = [("Soil␣Temperature", 0), ("Soil␣Moisture", 1), ("Soil␣EC", 2)]
    for label_text, _ in soil_temp_labels:
        frame = tk.Frame(soil_temp_frame, bg='lightyellow')
        frame.pack(fill=tk.X, padx=5, pady=5)
```

```
        label = tk.Label(frame, text=label_text+":␣", font=subtitle_font, bg='
    lightyellow')
        label.pack(side=tk.LEFT)

        value = tk.Label(frame, text="N/A", font=subtitle_font, bg='lightyellow')  #
    Placeholder for actual values
        value.pack(side=tk.LEFT, padx=5)

        unit = tk.Label(frame, text="", font=subtitle_font, bg='lightyellow')  #
    Placeholder for actual units
        unit.pack(side=tk.LEFT, padx=5)

        data_value_placeholders[label_text] = value
        data_unit_placeholders[label_text] = unit

    # Add content to leaf_frame
    leaf_title = tk.Label(leaf_frame, text="Leaf␣Data", font=subtitle_font, bg='
    lightcoral')
    leaf_title.pack(anchor='n')

    leaf_labels = [("Leaf␣Temperature", 0), ("Leaf␣Wetness", 1)]
    for label_text, _ in leaf_labels:
        frame = tk.Frame(leaf_frame, bg='lightcoral')
        frame.pack(fill=tk.X, padx=5, pady=5)

        label = tk.Label(frame, text=label_text+":␣", font=subtitle_font, bg='
    lightcoral')
        label.pack(side=tk.LEFT)

        value = tk.Label(frame, text="N/A", font=subtitle_font, bg='lightcoral')  #
    Placeholder for actual values
        value.pack(side=tk.LEFT, padx=5)

        unit = tk.Label(frame, text="", font=subtitle_font, bg='lightcoral')  #
    Placeholder for actual units
        unit.pack(side=tk.LEFT, padx=5)

        data_value_placeholders[label_text] = value
        data_unit_placeholders[label_text] = unit

    return root, main_frame, node_var, lat_lon_frame, soil_frame, env_frame,
    soil_temp_frame, leaf_frame, data_value_placeholders, data_unit_placeholders


# Function to update the GUI
def update_gui(data_value_placeholders, data_unit_placeholders, data_unitnode_var):
    while True:
        selected_site = node_var.get()
        if not ring_buffer_gui.is_empty():
            data = ring_buffer_gui.get()
            lat = data.get("lat")
            lon = data.get("lon")
            year = data.get("year")
            doy = data.get("doy")
            time = data.get("time")
            value = round(float(data.get("Value")),2)
            unit = data.get("Unit")
            if data.get("site_id") == selected_site:
```

```
                data_value_placeholders["Latitude"].config(text=f"{lat}")
                data_value_placeholders["Longitude"].config(text=f"{lon}")
                data_value_placeholders["Year"].config(text=f"{year}")
                data_value_placeholders["Date"].config(text=f"{doy}")
                data_value_placeholders["Time"].config(text=f"{time}")
                if data["Quantity"] == "pH":
                    data_value_placeholders["Soil pH"].config(text=f"{value}")
                    data_unit_placeholders["Soil pH"].config(text=f"{unit}")
                if data["Quantity"] == "Nitrogen":
                    data_value_placeholders["Soil Nitrogen"].config(text=f"{value}")
                    data_unit_placeholders["Soil Nitrogen"].config(text=f"{unit}")
                if data["Quantity"] == "Phosphorous":
                    data_value_placeholders["Soil Phosphorous"].config(text=f"{value}"
    )
                    data_unit_placeholders["Soil Phosphorous"].config(text=f"{unit}")
                if data["Quantity"] == "Potassium":
                    data_value_placeholders["Soil Potassium"].config(text=f"{value}")
                    data_unit_placeholders["Soil Potassium"].config(text=f"{unit}")
                if data["Quantity"] == "Rainfall":
                    data_value_placeholders["Rainfall"].config(text=f"{value}")
                    data_unit_placeholders["Rainfall"].config(text=f"{unit}")
                if data["Quantity"] == "Light":
                    data_value_placeholders["Light"].config(text=f"{value}")
                    data_unit_placeholders["Light"].config(text=f"{unit}")
                if data["Quantity"] == "Soil Temp":
                    data_value_placeholders["Soil Temperature"].config(text=f"{value}"
    )
                    data_unit_placeholders["Soil Temperature"].config(text=f"{unit}")
                if data["Quantity"] == "Soil Moisture":
                    data_value_placeholders["Soil Moisture"].config(text=f"{value}")
                    data_unit_placeholders["Soil Moisture"].config(text=f"{unit}")
                if data["Quantity"] == "Soil EC":
                    data_value_placeholders["Soil EC"].config(text=f"{value}")
                    data_unit_placeholders["Soil EC"].config(text=f"{unit}")
                if data["Quantity"] == "Leaf Temp":
                    data_value_placeholders["Leaf Temperature"].config(text=f"{value}"
    )
                    data_unit_placeholders["Leaf Temperature"].config(text=f"{unit}")
                if data["Quantity"] == "Leaf Wetness":
                    data_value_placeholders["Leaf Wetness"].config(text=f"{value}")
                    data_unit_placeholders["Leaf Wetness"].config(text=f"{unit}")
            ring_buffer_out.put(data)
```

4.5.5 | Data Transmission

After the GUI is updated, there is a thread that runs concurrently taking care of the data transmission to the cloud. This thread accesses data from the ring buffer **ring_bufffer_out**, and then parses it into a payload for an HTTPS POST request to send data to ThingSpeak as described in section 4.4. The following snippet shows the operation, that is also depicted in Figure 4.7.

```
### Do not execute the code as is!!!

def send_http_post(server_ip, port, message):
    http_request = (
        "POST /update HTTP/1.1\r\n"
        f"Host: {server_ip}\r\n"
        "Content-Type: application/x-www-form-urlencoded\r\n"
        f"Content-Length: {len(message)}\r\n\r\n"
```

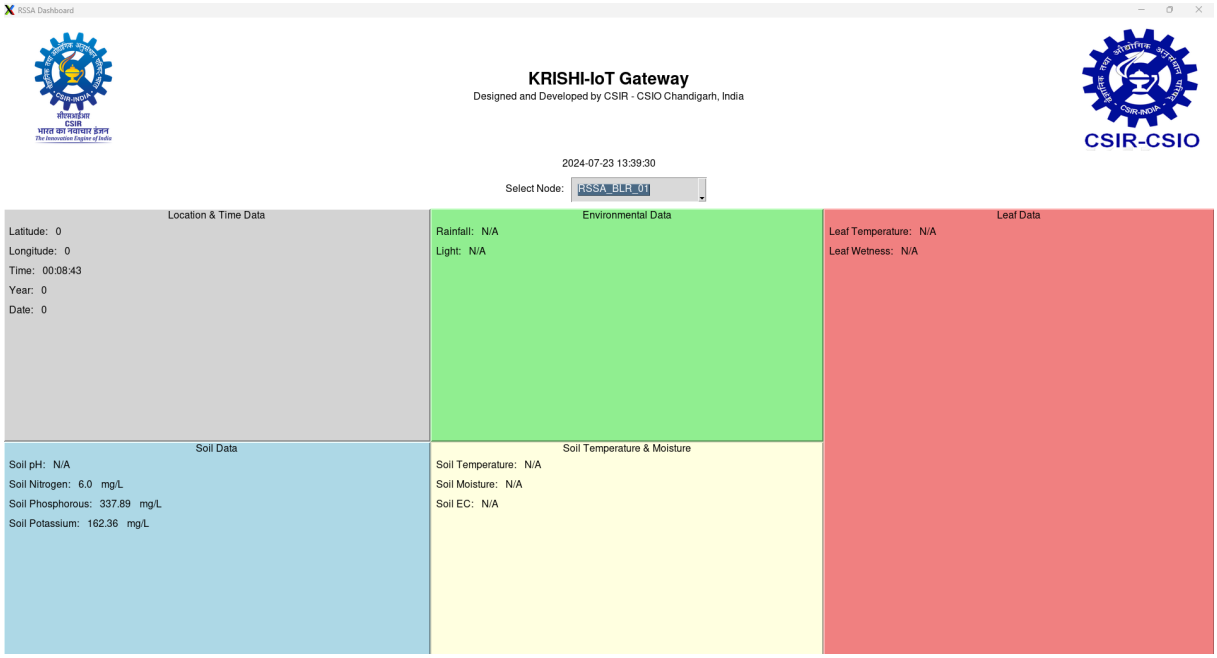

Figure 4.6: GUI after an update.

```
        f"{message}"
    )
    ser.write(http_request.encode())
    return send_at(b'\x1a'.decode(), 'OK', 5)

def send_to_thingspeak(data):

#    get_gps_position()
#    print(lat, lon)
    execute_gps_at_command('AT+CGPSINFO')
    print('GPS Executed')

    url = "https://api.thingspeak.com/update"

    payload = {
            'api_key': API_KEY,
            'field1': data.get("site_id"),  # Replace with your actual data fields
            'field2': data.get("lat"),  # Replace with your actual data fields
            'field3': data.get("lon"),  # Replace with your actual data fields
            'field4': data.get("year"),  # Replace with your actual data fields
            'field5': data.get("doy"),  # Replace with your actual data fields
            'field6': data.get("time"),  # Replace with your actual data fields
            'field7': data.get("Quantity"),  # Replace with your actual data fields
            'field8': str(round(float(data.get("Value")),2)),  # Replace with your
    actual data fields
            'latitude':str(round(lat, 4)),
            'longitude':str(round(lon, 4))
        }
    response = requests.post(url, data=payload)
    print("HTTP POST response:")
    print(response.status_code)
    print(response.text)

def update_cloud():
```

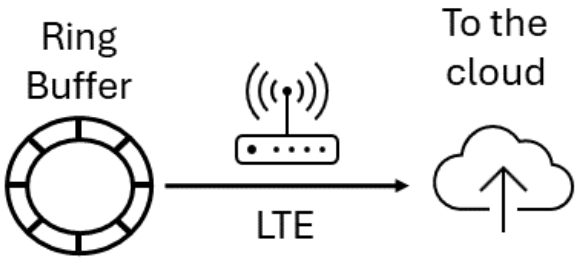

Figure 4.7: Flow of data during data transmission

```
    while True:
        if not ring_buffer_out.is_empty():
            data = ring_buffer_out.get()
            send_to_thingspeak(data)
            time.sleep(3)
```

4.5.6 | Final Code

All the sections of the software are now assembled into a single code. It starts off by first importing the requisite library for directory and file handling, HTTPS requests, Raspberry Pi GPIO, GUI, JSON objects, MQTT, multithreading and timing. This is followed by declaring and defining the macros for serial communication using UART for internet connection with the SIM7600X G-H hat. The macros for data transmission to ThingSpeak are also added. This is followed by the definition of functions that are attached to the different threads. This is followed by the main thread that initializes the SIM7600X G-H, takes the GPS fix, and then initializes and starts all the threads for the gateway to start working. The overall functionality of the gateway is shown in the Figure 4.8. The complete code for the gateway is in the following code block:

```
#RSSA Gateway Software
#Author - Swastik Bhattacharya
#Last Revision - 30th July 2024

#Import Libraries
import paho.mqtt.client as mqtt
import json
from collections import deque
import threading
import os
import json
import pandas as pd
import tkinter as tk
from tkinter import ttk
from tkinter import font as tkfont
import time
import psutil
from PIL import Image, ImageTk
from datetime import datetime
import serial
import RPi.GPIO as GPIO
import time
import os
import subprocess
import requests
import signal
import sys
```

```
#MQTT Macros
MQTT_ADDRESS = "192.168.4.1"
MQTT_USER = "rssa-gateway"
MQTT_PASSWORD = "rssa_gateway"

#Global variables
rec_buff=''
lat = 0
lon = 0

Latitude = ''; Longitude = ''

buff = ''

#Sites to display
sites = ["RSSA_CHD_01", "RSSA_BLR_01"]

#UI value and unit display placeholders
data_labels = ["Latitude", "Longitude", "Year", "Date", "Time", "Soil pH", "Soil
    Nitrogen", "Soil Phosphorous", "Soil Potassium",
                   "Rainfall", "Light", "Soil Temperature", "Soil Moisture", "Soil EC",
     "Leaf Temperature", "Leaf Wetness"]
data_units = ["", "", "", "", "", "", "mg/L", "mg/L", "mg/L", "mm", "lux", "C", "%", "
    uS/cm", "C", "%"]


class CircularQueue:
    """
    Class: CircularQueue
    ----------------------
    Create an object for a ring buffer

    Author:
        Swastik Bhattacharya.

    Date:
        30th July 2024.

    Functions:
        __init__ - constructor
        put - append to the buffer
        get - retrieve from the buffer
        is_empty - chack if the buffer is empty
    """

    def __init__(self, size):
        self.size = size
        self.buffer = [None] * size
        self.head = 0
        self.tail = 0
        self.count = 0
        self.lock = threading.Lock()

    def put(self, item):
        with self.lock:
            self.buffer[self.tail] = item
            self.tail = (self.tail + 1) % self.size
```

```
            if self.count == self.size:
                self.head = (self.head + 1) % self.size  # Overwrite the oldest data
            else:
                self.count += 1

    def get(self):
        with self.lock:
            if self.count == 0:
                return None
            item = self.buffer[self.head]
            self.head = (self.head + 1) % self.size
            self.count -= 1
            return item

    def is_empty(self):
        with self.lock:
            return self.count == 0

# Declare and initialize ring buffers for threads
RING_BUFFER_SIZE = 200
ring_buffer_in = CircularQueue(RING_BUFFER_SIZE)
ring_buffer_gui = CircularQueue(RING_BUFFER_SIZE)
ring_buffer_out = CircularQueue(RING_BUFFER_SIZE)

# GPIO and Serial Configuration for SIM7600X
ser = serial.Serial('/dev/ttyS0', 115200)
ser.flushInput()
power_key = 6
APN = 'airtelgprs.com'
ServerIP = 'api.thingspeak.com'
Port = '80'
#API_KEY = 'FQC043C665VOLO8Y'
API_KEY = '3MYY5DNQ1MD0091F'

def signal_handler(sig, frame):
    """
    Function: signal_handler
    ------------------------
    Handle the signals SIGINT and SIGTSTP

    Author:
        Swastik Bhattacharya.

    Date:
        30th July 2024
    """
    print("Signal received, closing the serial connection...")
    ser.close()
    #send_at('AT+CIPCLOSE=0', '+CIPCLOSE: 0,0', 15)
    #send_at('AT+NETCLOSE', '+NETCLOSE: 0', 1)
    GPIO.cleanup()
    power_down(power_key)
    kill_pppd()
    sys.exit(0)

# Register the signal handlers for Ctrl+C (SIGINT) and Ctrl+Z (SIGTSTP)
signal.signal(signal.SIGINT, signal_handler)
signal.signal(signal.SIGTSTP, signal_handler)
```

```
def power_on(power_key):
    """
    Function: power_on
    ----------------------
    Turn on the SIM7600X module

    Parameters:
        power_key : integer identifier for GPIO working as power key

    Returns:
        return_type: void

    Author:
        Swastik Bhattacharya.

    Date:
        30th July 2024.
    """
    print('SIM7600X is starting:')
    GPIO.setmode(GPIO.BCM)
    GPIO.setwarnings(False)
    GPIO.setup(power_key, GPIO.OUT)
    time.sleep(0.1)
    GPIO.output(power_key, GPIO.HIGH)
    time.sleep(2)
    GPIO.output(power_key, GPIO.LOW)
    time.sleep(20)
    print('SIM7600X is ready')

def power_down(power_key):
    """
    Function: power_down
    ----------------------
    Turn off the SIM7600X module

    Parameters:
        power_key : integer identifier for GPIO working as power key

    Returns:
        return_type: void

    Author:
        Swastik Bhattacharya.

    Date:
        30th July 2024.
    """
    print('SIM7600X is logging off:')
    GPIO.setmode(GPIO.BCM)
    GPIO.setup(power_key, GPIO.OUT)
    GPIO.setwarnings(False)
    GPIO.output(power_key, GPIO.HIGH)
    time.sleep(3)
    GPIO.output(power_key, GPIO.LOW)
    time.sleep(18)
    print('Goodbye')
```

```
def send_at(command, back, timeout):
    """
    Function: send_at
    ----------------------
    Send AT commands to the SIM7600X module

    Parameters:
        command : AT Command to be sent to the SIM7600X module
        back : What should the return look like
        timeout : Wait till timeout

    Returns:
        return_type: boolean

    Author:
        Swastik Bhattacharya.

    Date:
        30th July 2024.
    """
    global rec_buff
    rec_buff = ''
    ser.write((command + '\r\n').encode())
    time.sleep(timeout)
    if ser.inWaiting():
        time.sleep(0.1)
        rec_buff = ser.read(ser.inWaiting()).decode()
    if rec_buff:
        if back not in rec_buff:
            print(f"{command} ERROR")
            print(f"{command} response:\t{rec_buff}")
            return False
        else:
            print(rec_buff)
            return True
    else:
        print(f"{command} no response")
        return False

def send_http_post(server_ip, port, message):
    """
    Function: send_http_post
    ----------------------
    Send HTTP POST request using AT Command

    Parameters:
        server_ip : URL of the server to send the data
        port : Port through which the server is to be connected with
        message : Message to be sent

    Returns:
        return_type: boolean

    Author:
        Swastik Bhattacharya.

    Date:
        30th July 2024.
```

```
    """
    http_request = (
        "POST␣/update␣HTTP/1.1\r\n"
        f"Host:␣{server_ip}\r\n"
        "Content-Type:␣application/x-www-form-urlencoded\r\n"
        f"Content-Length:␣{len(message)}\r\n\r\n"
        f"{message}"
    )
    ser.write(http_request.encode())
    return send_at(b'\x1a'.decode(), 'OK', 5)

def send_to_thingspeak(data):
    """
    Function: send_to_thingspeak
    ----------------------
    Send the data as an HTTP POST request using requests library to thingspeak

    Parameters:
        data : JSON object to be sent

    Returns:
        return_type: void

    Author:
        Swastik Bhattacharya.

    Date:
        30th July 2024.
    """
    url = "https://api.thingspeak.com/update"
    payload = {
            'api_key': API_KEY,
            'field1': data.get("site_id"),  # Replace with your actual data fields
            'field2': data.get("lat"),  # Replace with your actual data fields
            'field3': data.get("lon"),  # Replace with your actual data fields
            'field4': data.get("year"),  # Replace with your actual data fields
            'field5': data.get("doy"),  # Replace with your actual data fields
            'field6': data.get("time"),  # Replace with your actual data fields
            'field7': data.get("Quantity"),  # Replace with your actual data fields
            'field8': data.get("Value"),  # Replace with your actual data fields
            'latitude':str(round(lat, 4)),
            'longitude':str(round(lon, 4))
        }
    response = requests.post(url, data=payload)
    print("HTTP␣POST␣response:")
    print(response.status_code)
    print(response.text)
    time.sleep(20)


# Function to set thread affinity
def set_thread_affinity(thread, core_id):
    """
    Function: set_thread_affinity
    ----------------------
    Set affinity of a thread to a core

    Parameters:
```

```
        thread : Thread to attach
        coure_id : The core index to which the thread is to be attached

    Returns:
        return_type: void

    Author:
        Swastik Bhattacharya.

    Date:
        30th July 2024.
    """
    p = psutil.Process(thread.ident)
    p.cpu_affinity([core_id])

#MQTT callback function for connection
def on_connect(client, userdata, flags, rc):
    """
    Function: on_connect
    ----------------------
    MQTT callback function for connection

    Returns:
        return_type: void

    Author:
        Swastik Bhattacharya.

    Date:
        30th July 2024.
    """
    print("Connected with Result Code", str(rc))
    client.subscribe("pH_data")
    client.subscribe("n_data")
    client.subscribe("p_data")
    client.subscribe("k_data")
    client.subscribe("rain_data")
    client.subscribe("light_data")
    client.subscribe("s_temp_data")
    client.subscribe("s_moist_data")
    client.subscribe("s_ec_data")
    client.subscribe("l_temp_data")
    client.subscribe("l_wet_data")

#MQTT callback function for messages
def on_message(client, userdata, msg):
    """
    Function: on_message
    ----------------------
    MQTT callback function for messages

    Returns:
        return_type: void

    Author:
        Swastik Bhattacharya.

    Date:
```

```
        30th July 2024.
    """
    print("Topic:", msg.topic)
    print("Message:", msg.payload.decode())
    payload = msg.payload.decode('utf-8')
    data = json.loads(msg.payload)
    ring_buffer_in.put(data)
    with open("gateway_data_rx.json", "a") as f:
        f.write(json.dumps(data) + "\n")

#MQTT start broker
def start_mqtt_broker():
    client = mqtt.Client()
    client.username_pw_set(MQTT_USER, MQTT_PASSWORD)
    client.on_connect = on_connect
    client.on_message = on_message
    client.connect("192.168.4.1", 1883)
    client.loop_forever()

def json_processing_thread():
    """
    Function: json_processing_thread
    ------------------------
    Thread target function to process JSON objects received

    Returns:
        return_type: void

    Author:
        Swastik Bhattacharya.

    Date:
        30th July 2024.
    """
    while True:
        json_obj = ring_buffer_in.get()
        if json_obj is not None:
            process_json(json_obj)
            #send_to_thingspeak(json_obj)
            ring_buffer_gui.put(json_obj)

def process_json(json_obj):
    """
    Function: process_json
    ------------------------
    Function to segregate and store the JSON objects

    Parameters:
        json_obj : The JSON object to be processed

    Returns:
        return_type: void

    Author:
        Swastik Bhattacharya.

    Date:
        30th July 2024.
```

```
    """
    site = json_obj['site_id']
    year = json_obj['year']
    doy = json_obj['doy']

    # Create directory path
    dir_path = os.path.expanduser(f"~/RSSA_data_log/{site}/{year}")
    os.makedirs(dir_path, exist_ok=True)

    # File path
    file_path = os.path.join(dir_path, f"{doy}.json")

    # Append JSON object to file
    with open(file_path, 'a') as f:
        f.write(json.dumps(json_obj) + "\n")

    #print(f"Processed and saved JSON: {json_obj}")

def update_time(label):
    """
    Function: update_time
    ---------------------
    Update current time in the GUI

    Parameters:
        label : Label to be updated for time

    Returns:
        return_type: void

    Author:
        Swastik Bhattacharya.

    Date:
        30th July 2024.
    """
    def update():
        current_time = datetime.now().strftime("%Y-%m-%d␣%H:%M:%S")
        label.config(text=current_time)
        label.after(1000, update)
    update()

def start_gui():
    """
    Function: start_gui
    -------------------
    Function to initialize the GUI

    Returns:
        root : The root of the tkinter tree making GUI
        main_frame : The main frame of the GUI
        node_var : Frame storing the receiving node drop-down
        lat_lon_frame : Frame to show Lat-Lon values of the node
        soil_frame : Frame to show the soil properties
        env_frame : Frame to show environmental parameters
        soil_temp_frame : Frame to show Soil TEM
        leaf_frame : Frame to show data from leaf sensor
        data_value_placeholders : Placeholders to show the received measurements
```

```
    data_unit_placeholders : Placeholders to show the units of the received
measurements

Author:
    Swastik Bhattacharya.

Date:
    30th July 2024.
"""
root = tk.Tk()
root.title("RSSA Dashboard")

# Maximize window
root.attributes('-zoomed', True)
root.configure(bg='white')  # Set background color to white

# Top Frame for logos and text
top_frame = tk.Frame(root, bg='white')
top_frame.pack(fill=tk.X)

left_logo = Image.open("CSIR_logo.png")  # Update with actual path
left_logo = left_logo.resize((200, 200), Image.ANTIALIAS)
left_logo = ImageTk.PhotoImage(left_logo)
left_logo_label = tk.Label(top_frame, image=left_logo, bg='white')
left_logo_label.image = left_logo
left_logo_label.pack(side=tk.LEFT, padx=10, pady=10)

right_logo = Image.open("CSIO_logo.png")  # Update with actual path
right_logo = right_logo.resize((200, 200), Image.ANTIALIAS)
right_logo = ImageTk.PhotoImage(right_logo)
right_logo_label = tk.Label(top_frame, image=right_logo, bg='white')
right_logo_label.image = right_logo
right_logo_label.pack(side=tk.RIGHT, padx=10, pady=10)

middle_frame = tk.Frame(top_frame, bg='white')
middle_frame.pack(expand=True)

title_font = tkfont.Font(family="Helvetica", size=40, weight="bold")
subtitle_font = tkfont.Font(family="Helvetica", size=32)

title_label = tk.Label(middle_frame, text="KRISHI-IoT Gateway", font=title_font,
bg='white')
title_label.pack()

subtitle_label = tk.Label(middle_frame, text="Designed and Developed by CSIR - 
CSIO Chandigarh, India", font=subtitle_font, bg='white')
subtitle_label.pack()

style = ttk.Style()

# Configure the style to add padding and change the font size
style.configure("TCombobox", padding=10, font=('Helvetica', 32))  # Adjust padding
 and font as needed

# Date and time
datetime_label = tk.Label(root, font=subtitle_font, bg='white')
datetime_label.pack()
update_time(datetime_label)
```

```
# Node selection
node_frame = tk.Frame(root, bg='white')
node_frame.pack(pady=10)

node_label = tk.Label(node_frame, text="Select␣Node:", font=subtitle_font, bg='
white')
node_label.pack(side=tk.LEFT)

node_var = tk.StringVar()
root.option_add("*TCombobox*Listbox*Font", subtitle_font)
node_dropdown = ttk.Combobox(node_frame, textvariable=node_var, state="readonly",
style="TCombobox", font = subtitle_font)
node_dropdown['values'] = ["RSSA_CHD_01", "RSSA_BLR_01"]  # Example values, update
 with actual site IDs
node_dropdown.pack(side=tk.LEFT, padx=10)

data_value_placeholders = {}; data_unit_placeholders = {}

# Main frame for the data sections
main_frame = tk.Frame(root, bg='white')
main_frame.pack(fill=tk.BOTH, expand=True)

# Create frames for five sections
lat_lon_frame = tk.Frame(main_frame, bg='lightgray', relief=tk.RAISED, bd=2)
soil_frame = tk.Frame(main_frame, bg='lightblue', relief=tk.RAISED, bd=2)
env_frame = tk.Frame(main_frame, bg='lightgreen', relief=tk.RAISED, bd=2)
soil_temp_frame = tk.Frame(main_frame, bg='lightyellow', relief=tk.RAISED, bd=2)
leaf_frame = tk.Frame(main_frame, bg='lightcoral', relief=tk.RAISED, bd=2)

# Use grid to place frames in a 3x2 pattern
lat_lon_frame.grid(row=0, column=0, sticky="nsew")
soil_frame.grid(row=1, column=0, sticky="nsew")
env_frame.grid(row=0, column=1, sticky="nsew")
soil_temp_frame.grid(row=1, column=1, sticky="nsew")
leaf_frame.grid(row=0, column=2, rowspan=2, sticky="nsew")

main_frame.grid_rowconfigure(0, weight=1)
main_frame.grid_rowconfigure(1, weight=1)
main_frame.grid_columnconfigure(0, weight=1)
main_frame.grid_columnconfigure(1, weight=1)
main_frame.grid_columnconfigure(2, weight=1)

# Add content to lat_lon_frame
lat_lon_title = tk.Label(lat_lon_frame, text="Location␣&␣Time␣Data", font=
subtitle_font, bg='lightgray')
lat_lon_title.pack(anchor='n')

lat_lon_labels = [("Latitude", 0), ("Longitude", 1), ("Time", 2), ("Year", 3), ("
Date", 4)]
for label_text, _ in lat_lon_labels:
    frame = tk.Frame(lat_lon_frame, bg='lightgray')
    frame.pack(fill=tk.X, padx=5, pady=5)

    label = tk.Label(frame, text=label_text+":␣", font=subtitle_font, bg='
lightgray')
    label.pack(side=tk.LEFT)
```

```
        value = tk.Label(frame, text="N/A", font=subtitle_font, bg='lightgray')  #
    Placeholder for actual values
        value.pack(side=tk.LEFT, padx=5)

        unit = tk.Label(frame, text="", font=subtitle_font, bg='lightgray')  #
    Placeholder for actual units
        unit.pack(side=tk.LEFT, padx=5)

        data_value_placeholders[label_text] = value
        data_unit_placeholders[label_text] = unit

    # Add content to soil_frame
    soil_title = tk.Label(soil_frame, text="Soil␣Data", font=subtitle_font, bg='
    lightblue')
    soil_title.pack(anchor='n')

    soil_labels = [("Soil␣pH", 0), ("Soil␣Nitrogen", 1), ("Soil␣Phosphorous", 2), ("
    Soil␣Potassium", 3)]
    for label_text, _ in soil_labels:
        frame = tk.Frame(soil_frame, bg='lightblue')
        frame.pack(fill=tk.X, padx=5, pady=5)

        label = tk.Label(frame, text=label_text+":␣", font=subtitle_font, bg='
    lightblue')
        label.pack(side=tk.LEFT)

        value = tk.Label(frame, text="N/A", font=subtitle_font, bg='lightblue')  #
    Placeholder for actual values
        value.pack(side=tk.LEFT, padx=5)

        unit = tk.Label(frame, text="", font=subtitle_font, bg='lightblue')  #
    Placeholder for actual units
        unit.pack(side=tk.LEFT, padx=5)

        data_value_placeholders[label_text] = value
        data_unit_placeholders[label_text] = unit

    # Add content to env_frame
    env_title = tk.Label(env_frame, text="Environmental␣Data", font=subtitle_font, bg='
    lightgreen')
    env_title.pack(anchor='n')

    env_labels = [("Rainfall", 0), ("Light", 1)]
    for label_text, _ in env_labels:
        frame = tk.Frame(env_frame, bg='lightgreen')
        frame.pack(fill=tk.X, padx=5, pady=5)

        label = tk.Label(frame, text=label_text+":␣", font=subtitle_font, bg='
    lightgreen')
        label.pack(side=tk.LEFT)

        value = tk.Label(frame, text="N/A", font=subtitle_font, bg='lightgreen')  #
    Placeholder for actual values
        value.pack(side=tk.LEFT, padx=5)

        unit = tk.Label(frame, text="", font=subtitle_font, bg='lightgreen')  #
    Placeholder for actual units
        unit.pack(side=tk.LEFT, padx=5)
```

```
        data_value_placeholders[label_text] = value
        data_unit_placeholders[label_text] = unit

    # Add content to soil_temp_frame
    soil_temp_title = tk.Label(soil_temp_frame, text="Soil␣Temperature␣&␣Moisture",
    font=subtitle_font, bg='lightyellow')
    soil_temp_title.pack(anchor='n')

    soil_temp_labels = [("Soil␣Temperature", 0), ("Soil␣Moisture", 1), ("Soil␣EC", 2)]
    for label_text, _ in soil_temp_labels:
        frame = tk.Frame(soil_temp_frame, bg='lightyellow')
        frame.pack(fill=tk.X, padx=5, pady=5)

        label = tk.Label(frame, text=label_text+":␣", font=subtitle_font, bg='
    lightyellow')
        label.pack(side=tk.LEFT)

        value = tk.Label(frame, text="N/A", font=subtitle_font, bg='lightyellow')  #
    Placeholder for actual values
        value.pack(side=tk.LEFT, padx=5)

        unit = tk.Label(frame, text="", font=subtitle_font, bg='lightyellow')  #
    Placeholder for actual units
        unit.pack(side=tk.LEFT, padx=5)

        data_value_placeholders[label_text] = value
        data_unit_placeholders[label_text] = unit

    # Add content to leaf_frame
    leaf_title = tk.Label(leaf_frame, text="Leaf␣Data", font=subtitle_font, bg='
    lightcoral')
    leaf_title.pack(anchor='n')

    leaf_labels = [("Leaf␣Temperature", 0), ("Leaf␣Wetness", 1)]
    for label_text, _ in leaf_labels:
        frame = tk.Frame(leaf_frame, bg='lightcoral')
        frame.pack(fill=tk.X, padx=5, pady=5)

        label = tk.Label(frame, text=label_text+":␣", font=subtitle_font, bg='
    lightcoral')
        label.pack(side=tk.LEFT)

        value = tk.Label(frame, text="N/A", font=subtitle_font, bg='lightcoral')  #
    Placeholder for actual values
        value.pack(side=tk.LEFT, padx=5)

        unit = tk.Label(frame, text="", font=subtitle_font, bg='lightcoral')  #
    Placeholder for actual units
        unit.pack(side=tk.LEFT, padx=5)

        data_value_placeholders[label_text] = value
        data_unit_placeholders[label_text] = unit

    return root, main_frame, node_var, lat_lon_frame, soil_frame, env_frame,
    soil_temp_frame, leaf_frame, data_value_placeholders, data_unit_placeholders

# Function to update the GUI
```

```
def update_gui(data_value_placeholders, data_unit_placeholders, data_unitnode_var):
    """
    Function: update_gui
    ----------------------
    Thread target function to update the GUI

    Parameters:
        data_value_placeholders : Placeholders to show received measurements
        data_unit_placeholders : Placeholders to show the units of the received
    measurements
        data_unitnode_var : Placeholder to store the site_id

    Returns:
        return_type : void

    Author:
        Swastik Bhattacharya.

    Date:
        30th July 2024.
    """
    while True:
        selected_site = node_var.get()
        if not ring_buffer_gui.is_empty():
            data = ring_buffer_gui.get()
            lat = data.get("lat")
            lon = data.get("lon")
            year = data.get("year")
            doy = data.get("doy")
            time = data.get("time")
            value = "N/A"
            if data.get("Value") or data.get("Value") == 0:
                value = round(float(data.get("Value")),2)
            unit = data.get("Unit")
            if data.get("site_id") == selected_site:
                data_value_placeholders["Latitude"].config(text=f"{lat}")
                data_value_placeholders["Longitude"].config(text=f"{lon}")
                data_value_placeholders["Year"].config(text=f"{year}")
                data_value_placeholders["Date"].config(text=f"{doy}")
                data_value_placeholders["Time"].config(text=f"{time}")
                if data["Quantity"] == "pH":
                    data_value_placeholders["Soil pH"].config(text=f"{value}")
                    data_unit_placeholders["Soil pH"].config(text=f"{unit}")
                if data["Quantity"] == "Nitrogen":
                    data_value_placeholders["Soil Nitrogen"].config(text=f"{value}")
                    data_unit_placeholders["Soil Nitrogen"].config(text=f"{unit}")
                if data["Quantity"] == "Phosphorous":
                    data_value_placeholders["Soil Phosphorous"].config(text=f"{value}"
    )
                    data_unit_placeholders["Soil Phosphorous"].config(text=f"{unit}")
                if data["Quantity"] == "Potassium":
                    data_value_placeholders["Soil Potassium"].config(text=f"{value}")
                    data_unit_placeholders["Soil Potassium"].config(text=f"{unit}")
                if data["Quantity"] == "Rainfall":
                    data_value_placeholders["Rainfall"].config(text=f"{value}")
                    data_unit_placeholders["Rainfall"].config(text=f"{unit}")
                if data["Quantity"] == "Light":
                    data_value_placeholders["Light"].config(text=f"{value}")
```

```
                    data_unit_placeholders["Light"].config(text=f"{unit}")
                if data["Quantity"] == "Soil Temp":
                    data_value_placeholders["Soil Temperature"].config(text=f"{value}"
    )
                    data_unit_placeholders["Soil Temperature"].config(text=f"{unit}")
                if data["Quantity"] == "Soil Moisture":
                    data_value_placeholders["Soil Moisture"].config(text=f"{value}")
                    data_unit_placeholders["Soil Moisture"].config(text=f"{unit}")
                if data["Quantity"] == "Soil EC":
                    data_value_placeholders["Soil EC"].config(text=f"{value}")
                    data_unit_placeholders["Soil EC"].config(text=f"{unit}")
                if data["Quantity"] == "Leaf Temp":
                    data_value_placeholders["Leaf Temperature"].config(text=f"{value}"
    )
                    data_unit_placeholders["Leaf Temperature"].config(text=f"{unit}")
                if data["Quantity"] == "Leaf Wetness":
                    data_value_placeholders["Leaf Wetness"].config(text=f"{value}")
                    data_unit_placeholders["Leaf Wetness"].config(text=f"{unit}")
            ring_buffer_out.put(data)

def update_cloud():
    """
    Function: update_cloud
    ----------------------
    Thread target function to update the ThingSpeak cloud

    Returns:
        return_type : void

    Author:
        Swastik Bhattacharya.

    Date:
        30th July 2024.
    """
    while True:
        if not ring_buffer_out.is_empty():
            data = ring_buffer_out.get()
            send_to_thingspeak(data)

def get_gps_position():
    """
    Function: get_gps_position
    ----------------------
    Get the GPS position fix

    Returns:
        return_type : void

    Author:
        Swastik Bhattacharya.

    Date:
        30th July 2024.
    """
    rec_null = True
    global buff, rec_buff, lat, lon;
    answer = 0
```

```
    print('Start␣GPS␣session...')
    rec_buff = ''
    send_at('AT+CGPS=1,1','OK',5)
    time.sleep(2)
    while rec_null:
        answer = send_at('AT+CGPSINFO','+CGPSINFO:␣',5)
        if 1 == answer:
            answer = 0
            if ',,,,,,' in rec_buff:
                print('GPS␣is␣not␣ready')
#                rec_null = False
                time.sleep(1)
            elif 'AT+CGPSINFO' in rec_buff:
#                print(rec_buff)
#                break
                buff = rec_buff[25:]
#                print(len(buff))
#                buff = buff[1:][11:]
                print(buff)
                gps_info = buff.split(',')
                lat = float(gps_info[0])//100 + float(gps_info[0][2:])/60;
                lon = float(gps_info[2])//100 + float(gps_info[2][3:])/60
                if gps_info[1] == 'S':
                    lat *= -1
                if gps_info[3] == 'W':
                    lon *= -1
                Latitude = str(lat); Longitude = str(lon);
                rec_null = False
                break
        else:
            print('error␣%d'%answer)
            rec_buff = ''
            send_at('AT+CGPS=0','OK',1)
            return False
        time.sleep(1.5)

def kill_pppd():
    """
    Function: kill_pppd
    -----------------------
    Kill the pppd process when it interferes with ops

    Returns:
        return_type : void

    Author:
        Swastik Bhattacharya.

    Date:
        30th July 2024.
    """
    try:
        # Get the list of all running processes
        processes = subprocess.check_output(['ps', 'aux'])
        processes = processes.decode('utf-8').splitlines()

        # Search for the pppd process
        for process in processes:
```

```
            if 'pppd' in process:
                # Split the process info to get the PID (second column)
                parts = process.split()
                pid = int(parts[1])
                print(f'Killing pppd process with PID: {pid}')
                # Kill the process
                print(pid)
                os.kill(pid, 9)
                print(f'Process {pid} killed.')
                return
        print('pppd process not found.')

    except Exception as e:
        print(f'An error occurred: {e}')


# Main function
if __name__ == "__main__":

    try:
        GPIO.setmode(GPIO.BCM)  # Set the GPIO mode at the start
        kill_pppd()
        time.sleep(5)

#        power_down(power_key)

        power_on(power_key)

        get_gps_position()

    # Ensure PPPD is installed
#    os.system('sudo apt-get update')
#    os.system('sudo apt-get install -y ppp')

    # Create PPP chat script
        chat_script = '''TIMEOUT 3
ABORT '\\nBUSY\\r'
ABORT '\\nNO ANSWER\\r'
ABORT '\\nRINGING\\r\\n\\r\\nRINGING\\r'
ABORT '\\nNO CARRIER\\r'
ABORT '\\nNO DIALTONE\\r'
ABORT '\\nNO DIAL TONE\\r'
ABORT '\\nNO\\(\\s\\)DIAL TONE\\r'
ABORT '\\nNO CONNECT\\r'
'' AT
OK ATH
OK ATE1
OK AT+CGDCONT=1,"IP","airtelgprs.com"
OK ATD*99#
CONNECT '''.strip()

        with open('/etc/chatscripts/gprs', 'w') as f:
            f.write(chat_script)

    # Create PPP peers file
        ppp_peers = '''/dev/ttyS0 115200
connect "/usr/sbin/chat -v -f /etc/chatscripts/gprs"
noauth
```

```
defaultroute
usepeerdns
persist
'''.strip()

        with open('/etc/ppp/peers/gprs', 'w') as f:
            f.write(ppp_peers)

    # Start PPP connection
        subprocess.run(['sudo', 'pppd', 'call', 'gprs'])

    # Wait to ensure connection is established
        time.sleep(10)

    # Print IP address and DNS information
        ip_info = subprocess.check_output("ifconfig␣ppp0", shell=True).decode()
        print("PPP0␣interface␣info:")
        print(ip_info)

    # Check /etc/resolv.conf for DNS servers
        resolv_conf = subprocess.check_output("cat␣/etc/resolv.conf", shell=True).
    decode()
        print("DNS␣settings␣in␣/etc/resolv.conf:")
        print(resolv_conf)

    # Test the connection by pinging a server
        ping_response = os.system("ping␣-c␣4␣google.com")

        if ping_response == 0:
            print("Internet␣connection␣established.")
        else:
            print("Failed␣to␣establish␣internet␣connection.")

        root, main_frame, node_var, lat_lon_frame, soil_frame, env_frame,
    soil_temp_frame, leaf_frame, data_value_placeholders, data_unit_placeholders =
    start_gui()

        mqtt_thread = threading.Thread(target=start_mqtt_broker)
        set_thread_affinity(mqtt_thread, 0)
        processing_thread = threading.Thread(target=json_processing_thread)
        set_thread_affinity(processing_thread, 1)
        gui_update_thread = threading.Thread(target=update_gui, args=(
    data_value_placeholders, data_unit_placeholders, node_var))
        set_thread_affinity(gui_update_thread, 2)
        send_http_thread = threading.Thread(target=update_cloud)
        set_thread_affinity(send_http_thread, 3)

        mqtt_thread.start()
        processing_thread.start()
        gui_update_thread.start()
        send_http_thread.start()

        root.mainloop()

        mqtt_thread.join()
        processing_thread.join()
        gui_update_thread.join()
        send_http_thread.join()
```

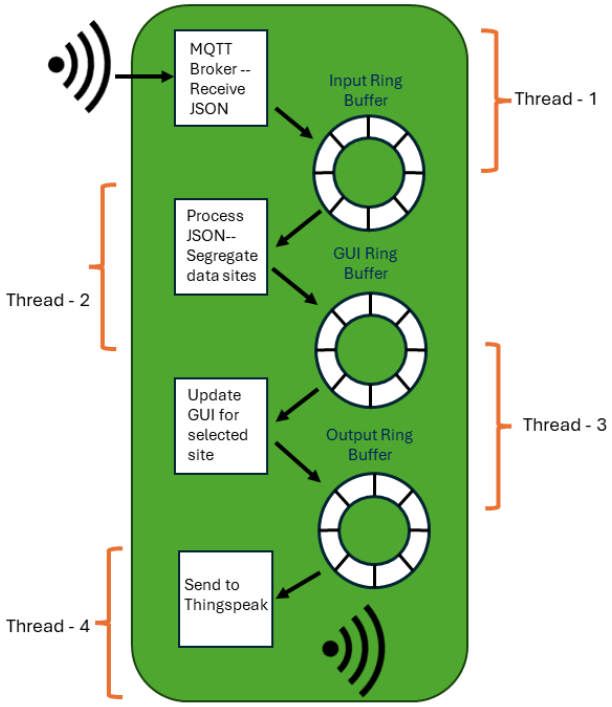

Figure 4.8: Complete gateway operations.

```
    except Exception as e:
        print(f"An␣error␣occurred:␣{e}")
#        send_at('AT+CIPCLOSE=0', '+CIPCLOSE: 0,0', 15)
#        send_at('AT+NETCLOSE', '+NETCLOSE: 0', 1)
        GPIO.cleanup()
        power_down(power_key)
        ser.close()
        kill_pppd()
        sys.exit(1)
```

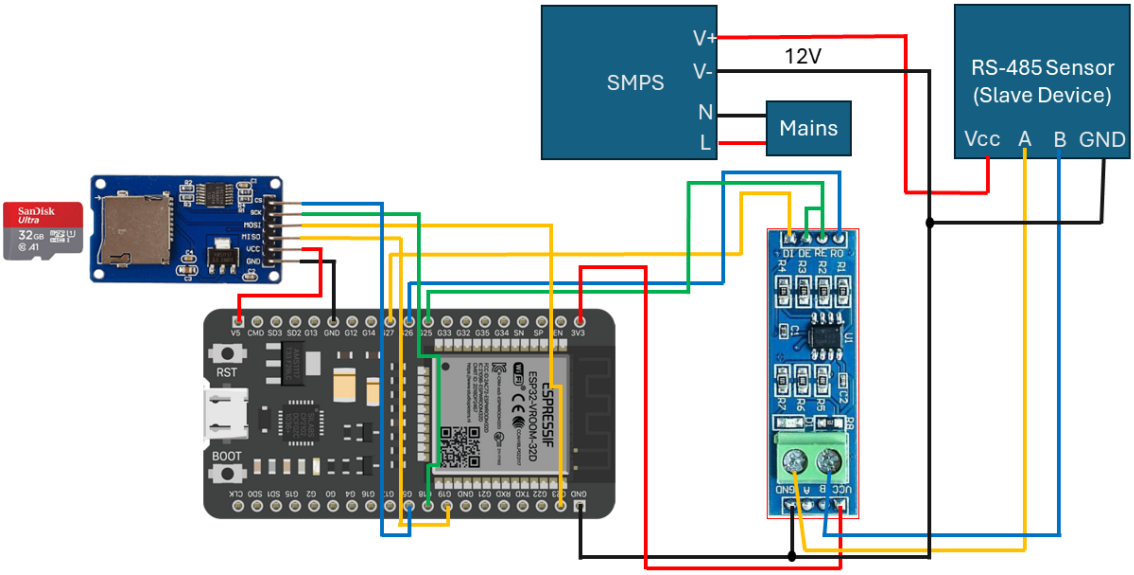

Figure 5.1: Schematic of sensor and peripheral interfacing to test ESP32 node datalogger.

5 | Demonstration

5.1 | Node Demonstration

The experimentation of the node and the gateway hardwares have involved several steps. Initially, the ideation of the node has been tested using a simple bread-board model implemented as per Figure 5.1. The core hardware used is the ESP32 Node MCU. For initial testing, it is connected to the GPS module, SD card module, an SMPS, a MAX485 hardware, and a few RS485 sensors so measure incident light, rainfall, soil moisture, soil EC, soil temperature, and soil NPK.

Once this breadboard model is turned on, it starts acquiring data from the sensors with the help of the code in Section 3.5.4. The data is stored into the attached SD card using the SD card module interfaced with the ESP32 NodeMCU. The model first takes a GPS fix, and then uses it to put time stamps and location information into the JSON objects to record the sensor data. The experiment setup when the sensors are installed is shown Figure 5.2.

The breadboard setup is kept active for about 7 hours. The data acquired is then analyzed to check its integrity. This is checked as a rule of thumb, since there were no standardized samples available at the time of testing. The testing environment was under a shade of lot of trees, therefore the light sensor could receive only speckles of light at different time intervals. This is shown from the plotted measurements in Figure 5.3. Since there has been no change in the property of the soil, the parameters of the soil will be remaining constant except a few outliers that could be introduced due to some disturbance such as insects stepping on sensor probes, etc. This can interfere with the sensing mechanism at certain points of time. This is also corroborated by the plot in Figure 5.4 for soil Nitrogen.

After the testing of the breadboard model, the node hardware PCB is fabricated and tested. The fabricated PCB is kept in an enclosure, whose design drawings are in Appendix B. The enclosure has 3 main components: The main box, a bracket, and a cover. The main box has a raised spacer to fix the PCB such that the LoRa SMA port could be protruding out of the box through a provided incision. An SMPS could be installed inside the box. There are incisions provided for a power socket that can be connected to the SMPS for a connection to the main power supply. There are also incisions provided at both sides to connect 4 RS485 sensors. This assembly is shown in Figure 5.5. The bracket inside the enclosure is to affix GPS module and an OLED display. After this, a cover is put on the box and bracket. The complete assembly is shown in Figure 5.6. On the box, there are also incisions to have SMA ports for WiFi connection and GPS antenna.

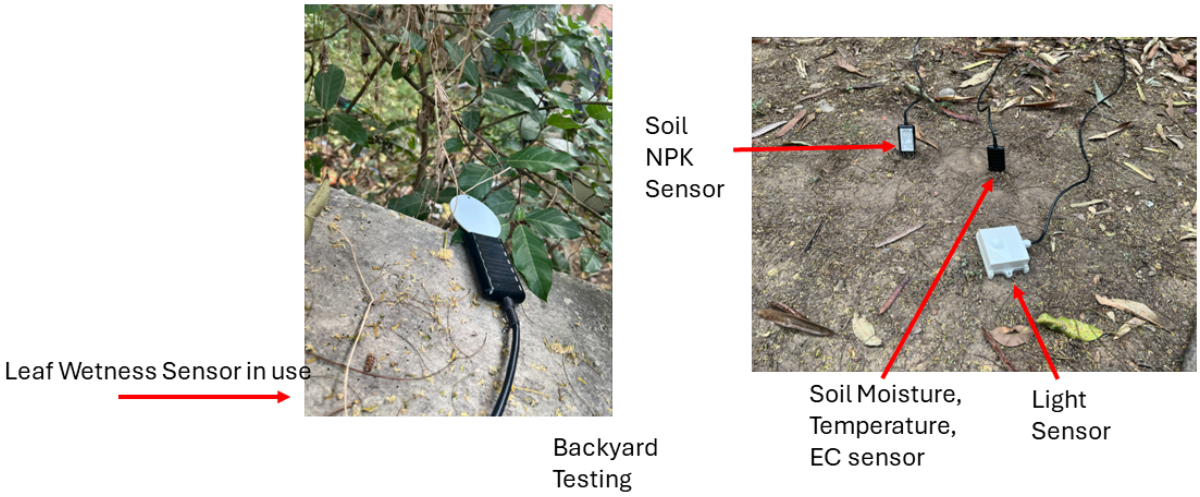

Figure 5.2: Sensors set up in the field to test the node datalogger.

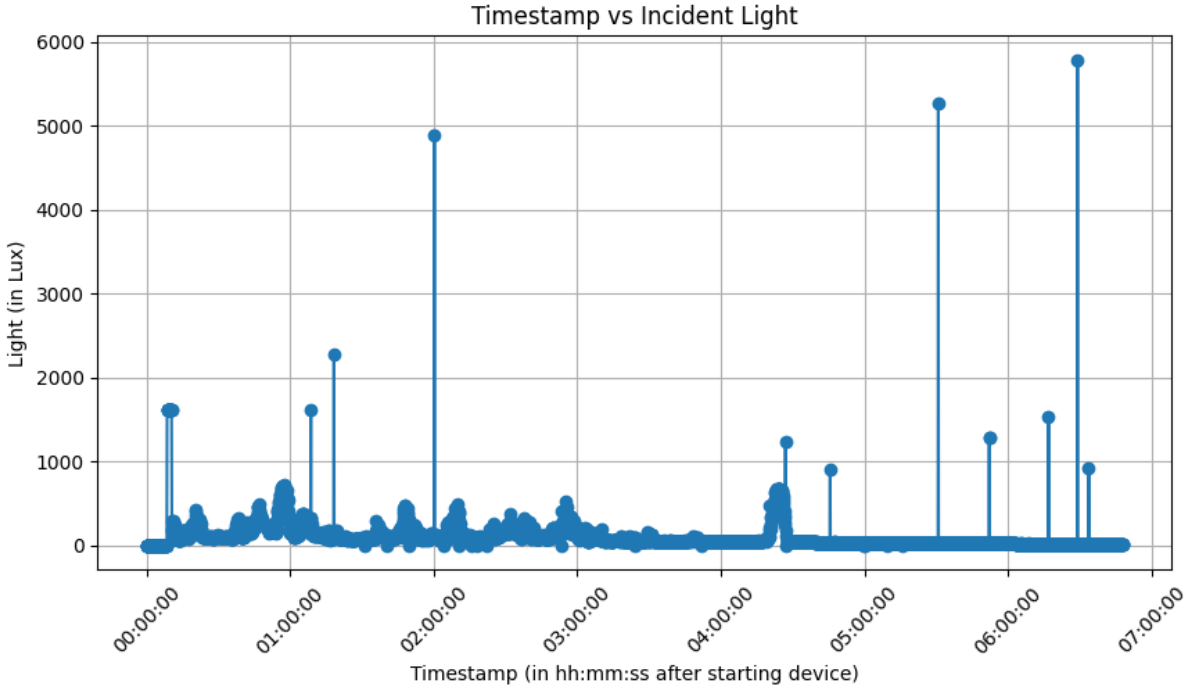

Figure 5.3: The trend of measurements taken of incoming sunlight.

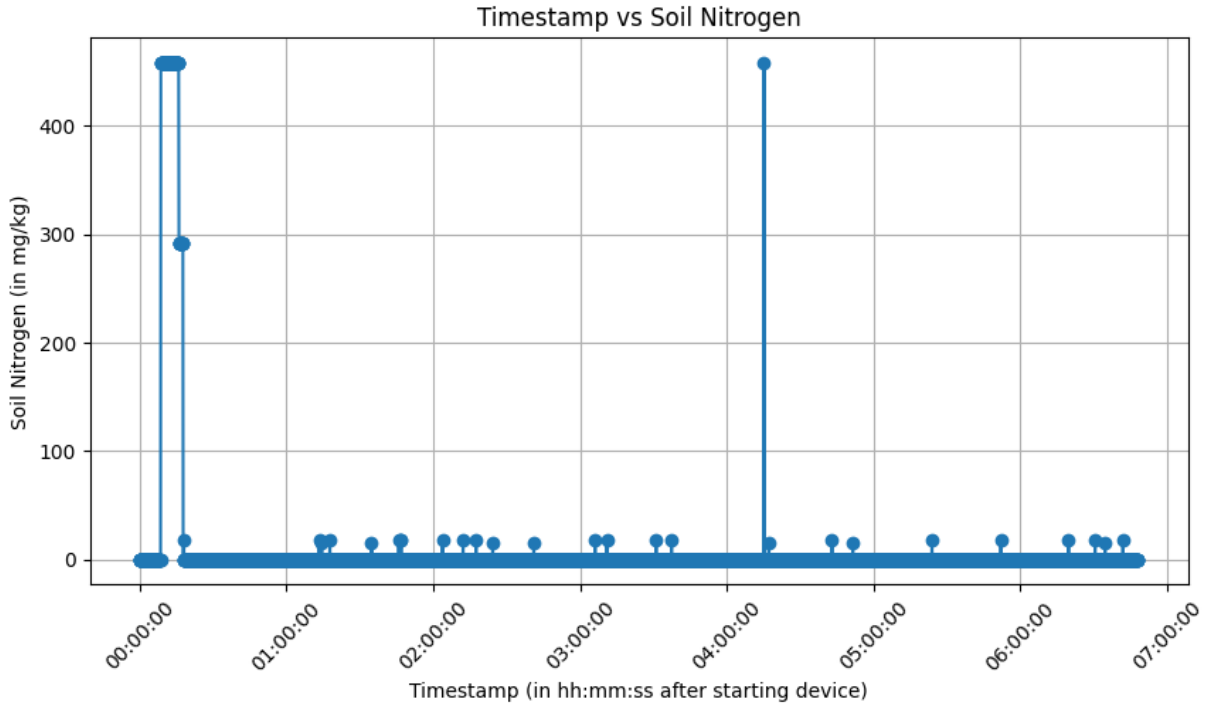

Figure 5.4: The trend of measurements taken of soil nitrogen over about 7 hours.

5.2 | Gateway Demonstration

The gateway is developed using a Raspberry Pi 4B. The gateway is assembled by first connecting the Raspberry Pi to 5.5 inch Waveshare LCD display using the microHDMI port on the Raspberry Pi. For the touch control, there is a separate microUSB B port on the display that can be connected to the USB port of the Raspberry Pi. Then, the SIM7600X G-H hat is affixed to the Raspberry Pi with the help of the pogo pin connectors of the Raspberry Pi and the SIM7600X G-H hat. There could be need for the use of pogo pin extenders and spacers in case one would need to use a fan for cooling purposes, as in the case here. Appendix C consists of the drawings for the enclosure for the gateway. There are four components: the main enclosure box, two vents, and a back cover. The main enclousure houses the gateway setup, with incisions for type-C USB connector for power supply, and two SMA antenna outlets for the LTE and GPS antennas for the SIM760X G-H module. The vents are capable to have fans attached to them and are affixed on the sides of the main enclosure. The back cover has a wedge that supports the assembly so that its position stabilizes. After the assembly, the gateway device is shown in Figure 5.7. During the assembly, it is assumed that the Raspberry Pi is configured to be used as a WiFi access point.
Once the gateway is activated, with the help of a USB keyboard and mouse, the script for the gateway software is executed and if there is an active node nearby transmitting data, the gateway captures it, stores it, updates the GUI, and transmits it to the ThingSpeak cloud. The ThingSpeak has capability to send data only once in every 15 seconds. This limits the data rate at which the gateway will be transmitting the data to the cloud. However, this acts as a good litmus test to check whether the gateway is good enough to transmit data. The data sent to the ThingSpeak cloud can be downloaded by the user as a Comma Separated Variable (CSV) file. Figure 5.8 shows a CSV file example that has recorded the measurements taken by the node. Figure 5.9 shows an example setup of gateway and node hardwares.

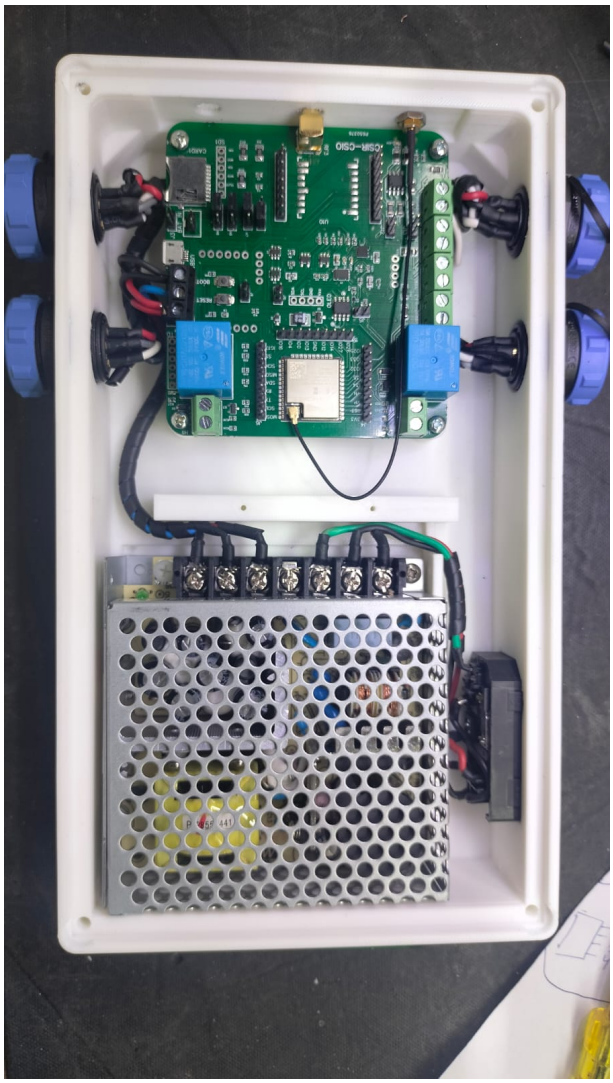

Figure 5.5: Wire Assembly of the node hardware.

Figure 5.6: Assembled Node Hardware

Figure 5.7: Assembled gateway device.

	A	B	C	D	E	F	G	H	I	J	K	L
1	created_at	entry_id	field1	field2	field3	field4	field5	field6	field7	field8	latitude	longitude
1970	2024-06-25T12:50:10+00:00	1969	RSSA_BLR_01	0	0	0	0	03:14:51	Phosphorous	616.8		
1971	2024-06-25T12:51:32+00:00	1970	RSSA_BLR_01	0	0	0	0	03:16:47	Nitrogen	575.84		
1972	2024-06-25T12:53:53+00:00	1971	RSSA_BLR_01	0	0	0	0	03:19:09	Nitrogen	575.84		
1973	2024-06-25T12:55:19+00:00	1972	RSSA_BLR_01	0	0	0	0	03:20:35	Nitrogen	575.84		
1974	2024-06-25T12:55:53+00:00	1973	RSSA_BLR_01	0	0	0	0	03:20:35	Phosphorous	616.8		
1975	2024-06-25T12:57:15+00:00	1974	RSSA_BLR_01	0	0	0	0	03:22:30	Nitrogen	575.84		
1976	2024-06-25T13:03:44+00:00	1975	RSSA_BLR_01	0	0	0	0	03:28:59	Nitrogen	575.84		
1977	2024-06-25T13:08:57+00:00	1976	RSSA_BLR_01	0	0	0	0	03:34:13	Nitrogen	575.84		
1978	2024-06-25T13:09:31+00:00	1977	RSSA_BLR_01	0	0	0	0	03:34:13	Phosphorous	616.8		
1979	2024-06-25T13:18:10+00:00	1978	RSSA_BLR_01	0	0	0	0	03:43:26	Nitrogen	575.84		
1980	2024-06-25T13:19:32+00:00	1979	RSSA_BLR_01	0	0	0	0	03:44:48	Nitrogen	616.8		
1981	2024-06-26T07:01:48+00:00	1980	RSSA_BLR_01	0	0	0	0	00:00:01	Nitrogen	424.05		
1982	2024-06-26T07:03:07+00:00	1981	RSSA_BLR_01	0	0	0	0	00:01:19	Nitrogen	616.8		
1983	2024-06-26T07:03:42+00:00	1982	RSSA_BLR_01	0	0	0	0	00:01:19	Phosphorous	616.05		
1984	2024-06-26T07:04:49+00:00	1983	RSSA_BLR_01	0	0	0	0	00:01:23	Nitrogen	616.8		
1985	2024-06-26T07:05:23+00:00	1984	RSSA_BLR_01	0	0	0	0	00:01:23	Phosphorous	616.05		
1986	2024-06-26T07:05:58+00:00	1985	RSSA_BLR_01	0	0	0	0	00:01:23	Potassium	424.05		
1987	2024-06-26T07:06:32+00:00	1986	RSSA_BLR_01	0	0	0	0	00:01:27	Nitrogen	248.16		
1988	2024-06-26T07:07:58+00:00	1987	RSSA_BLR_01	0	0	0	0	00:06:10	Nitrogen	616.8	30.7127	76.7838
1989	2024-06-26T07:08:32+00:00	1988	RSSA_BLR_01	0	0	0	0	00:06:10	Phosphorous	616.05	30.7127	76.7838
1990	2024-06-26T07:10:36+00:00	1989	RSSA_BLR_01	0	0	0	0	00:08:14	Phosphorous	616.05	30.7127	76.7838
1991	2024-06-26T10:20:31+00:00	1990	RSSA_BLR_01	0	0	0	0	03:19:08	Nitrogen	616.8	30.7128	76.7838
1992	2024-06-26T10:24:34+00:00	1991	RSSA_BLR_01	0	0	0	0	03:22:15	Phosphorous	616.8	30.7129	76.7838
1993	2024-06-26T10:25:07+00:00	1992	RSSA_BLR_01	0	0	0	0	03:22:15	Potassium	162.52	30.7129	76.7838
1994	2024-06-26T10:25:39+00:00	1993	RSSA_BLR_01	0	0	0	0	03:22:19	Nitrogen	616.8	30.713	76.7838
1995	2024-06-26T10:26:11+00:00	1994	RSSA_BLR_01	0	0	0	0	03:22:19	Phosphorous	616.8	30.713	76.7838
1996	2024-06-26T10:26:43+00:00	1995	RSSA_BLR_01	0	0	0	0	03:22:19	Potassium	162.52	30.7128	76.7838
1997	2024-06-26T10:27:16+00:00	1996	RSSA_BLR_01	0	0	0	0	03:22:22	Nitrogen	616.8	30.7128	76.7838
1998	2024-06-26T10:27:48+00:00	1997	RSSA_BLR_01	0	0	0	0	03:22:22	Phosphorous	616.8	30.7128	76.7838
1999	2024-06-26T10:28:20+00:00	1998	RSSA_BLR_01	0	0	0	0	03:22:22	Potassium	162.52	30.7129	76.7838
2000	2024-06-26T10:28:52+00:00	1999	RSSA_BLR_01	0	0	0	0	03:22:56	Phosphorous	616.8	30.7128	76.7838
2001	2024-06-26T10:29:24+00:00	2000	RSSA_BLR_01	0	0	0	0	03:23:29	Phosphorous	616.8	30.7128	76.7838
2002	2024-06-26T10:29:57+00:00	2001	RSSA_BLR_01	0	0	0	0	03:23:59	Phosphorous	616.8	30.7128	76.7838

Figure 5.8: ThingSpeak CSV Example

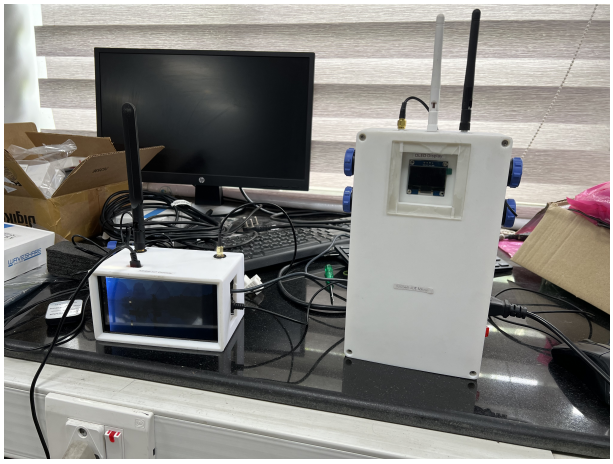

Figure 5.9: The setup of the gateway and node together

6 | Future Scope

Driven by the urgent need for sustainable farming techniques and technology breakthroughs, the future of IoT in smart agriculture is primed for revolutionary growth. Combining machine learning (ML) and artificial intelligence (AI) with Internet of Things (IoT) devices is one of the biggest developments. Predictive analytics is made possible by these technologies, which give farmers the ability to foresee and reduce risks like pest infestations, weather fluctuations, and crop illnesses. AI and ML can analyse massive volumes of data gathered from IoT devices to produce actionable insights that help improve crop output, optimise irrigation schedules, and cut down on resource waste. To ensure maximum production and quality, AI systems, for instance, can determine the best time to harvest a crop based on weather forecasts and crop growth patterns.

Using drone technology in conjunction with IoT networks is another new trend. Large agricultural areas can be surveyed by drones fitted with multispectral sensors and cameras, which can then provide high-resolution information on crop health, soil conditions, and moisture levels. Farmers can receive this data in real-time through IoT platforms, allowing for targeted and efficient actions. Furthermore, planting, pesticide application, and animal monitoring can be programmed into autonomous drones, which further boosts agricultural operations' production and efficiency. Drones, for example, can precisely apply more water or nutrients to fields that require it, cutting down on waste and encouraging consistent crop development. Currently, CSIR-CIMAP is undertaking airborne campaigns using drones developed and provided by CSIR-NAL with multispectral satellites as payloads for insights into soil conditions and crop health.

IoT applications in smart agriculture are about to undergo a revolution thanks to the widespread use of 5G technology. 5G will enable the deployment of more IoT devices across large farmlands, improving real-time data collecting and analysis, thanks to its high-speed connectivity and low latency. Advanced applications like remote machinery operation, augmented reality for training and support, and improved food product traceability from farm to fork will also be supported by this connectivity. An agricultural ecosystem that is more responsive and integrated will be made possible by the seamless connectivity of more devices. For instance, farmers will be able to remotely monitor and manage irrigation systems, guaranteeing efficient water use and cutting labour expenses.

Another area where blockchain technology is gaining interest is smart agriculture. Blockchain technology has the potential to improve agricultural supply networks' transparency and traceability by offering a secure and decentralised method of recording transactions and data. In order to ensure food safety and authenticity, farmers can utilise blockchain to trace the origin, quality, and movement of their produce. In response, customers can validate a product's sustainability and provenance by scanning the QR codes on it. Additionally, this technology can help create agricultural insurance and financing systems that are more open and efficient.

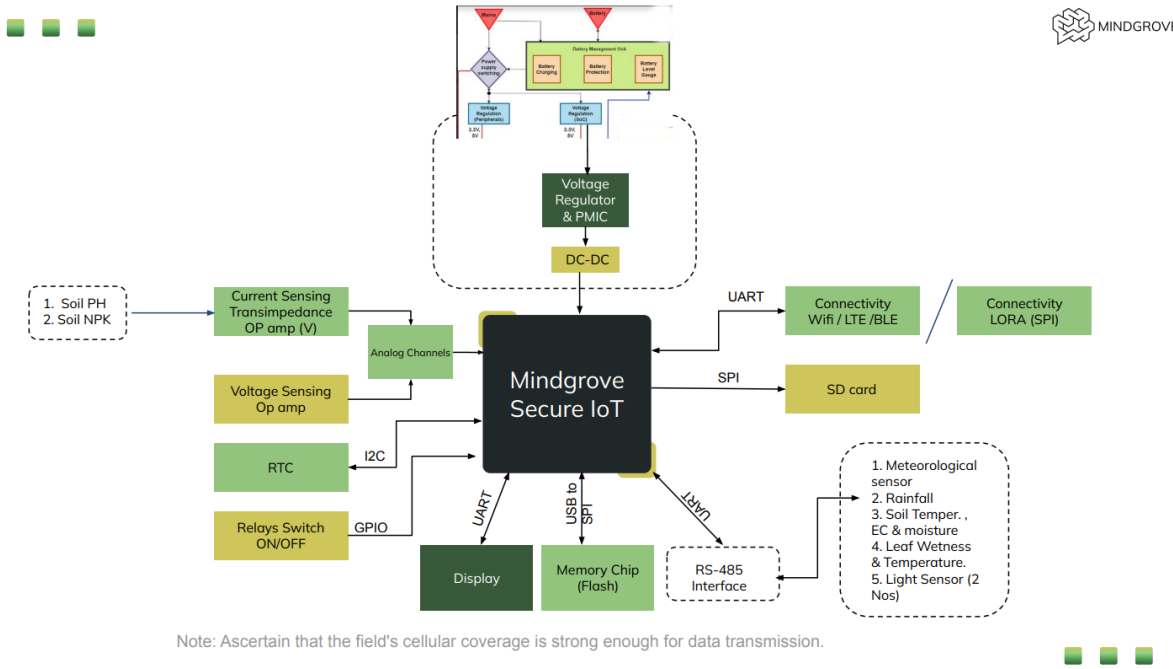

Figure 6.1: Proposed block diagram of the node by Mindgrove Technologies.

In the future, sustainable agriculture techniques and environmental effect will be essential components. Precision agriculture is made possible by IoT technology, which is essential to the advancement of sustainable agriculture. Farmers may minimise their influence on the environment by using IoT sensors to monitor soil health, optimise water usage, and limit the application of fertilisers and pesticides. Furthermore, by offering in-depth insights about crop performance and soil health, IoT systems can assist regenerative agricultural techniques like crop rotation and cover crops. These methods aid in the fight against climate change by preserving soil fertility, promoting biodiversity, and sequestering carbon. To facilitate this, CSIR-CSIO is developing an IoT network and datalogger system. The development process has been discussed at length in this report. Keeping in mind various issues related to availability of semiconductor components due to current supply chains, there is an effort by CSIR-CSIO to develop an indigenous IoT datalogger and gateway using an SoC based on the Shakti microcontroller in collaboration with Mindgrove Technologies, a startup incubated by the Indian Institute of Technology - Madras, Chennai. Figure 6.1 shows the initial block diagram for the datalogger being co-developed with Mindgrove.
Another interesting usage of IoT in agriculture is smart greenhouses. These controlled environments create ideal growing conditions for plants by controlling temperature, humidity, light, and CO2 levels using IoT sensors and automation systems. Higher yields and year-round production are made possible by this technique, independent of the outside weather. Furthermore, by precisely controlling and monitoring their usage, smart greenhouses can lower their energy and water consumption, increasing their sustainability and economic viability. CSIR-CEERI is currently developing an CEA as part of the project of this work report.

7 | References

[1] 5.5inch 1440x2560 lcd - waveshare wiki.

[2] Max485 datasheet and product info.

[3] Raspberry pi documentation - configuration.

[4] Sim7600e-h 4g hat - waveshare wiki.

[5] Sabrina Akhtar. Integrated iot (internet of things) system solution for smart agriculture management, July 28 2020. US Patent 10,728,336.

[6] David Astély, Erik Dahlman, Anders Furuskär, Ylva Jading, Magnus Lindström, and Stefan Parkvall. Lte: the evolution of mobile broadband. *IEEE Communications magazine*, 47(4):44–51, 2009.

[7] Massimo Banzi and Michael Shiloh. *Getting Started with Arduino: The Open Source Electronics Prototyping Platform*. Maker Media, Inc., 2020.

[8] Michael Bender, Gautam K Bhat, Rhonda L Childress, and Nalini Muthurajan. Cognitive based decision support system for agriculture, February 22 2022. US Patent 11,257,172.

[9] Tim Berners-Lee and Robert Cailliau. *World Wide Web: A Visionary Guide*. Addison-Wesley, Reading, MA, 1999. A guide to the origins and evolution of the World Wide Web, including HTTP.

[10] David R. Butenhof. *Programming with POSIX threads*. Addison-Wesley Longman Publishing Co., Inc., USA, 1997.

[11] Wikipedia contributors. Agriculture in india, 2023. Accessed: 2023-07-27.

[12] Diane J. Cook and Sajal K. Das, editors. *Wireless Sensor Networks: Smart Environments and Technologies*. John Wiley Sons, 2007.

[13] Brian P Crow, Indra Widjaja, Jeong Geun Kim, and Prescott T Sakai. Ieee 802.11 wireless local area networks. *IEEE Communications magazine*, 35(9):116–126, 1997.

[14] Prabal Dutta and David Culler. System software techniques for low-power operation in wireless sensor networks. In *Proceedings of the 2008 IEEE/ACM International Conference on Computer-Aided Design*, pages 725–732. IEEE, 2008.

[15] Eclipse. Eclipse/mosquitto: Eclipse mosquitto - an open source mqtt broker.

[16] Food and Agriculture Organization. *FAO Statistical Yearbook*. Food and Agriculture Organization of the United Nations, 2013. Accessed: 2023-07-27.

[17] Food and Agriculture Organization. Good agricultural practices: A working concept, 2003. Accessed: 2023-07-27.

[18] Behrouz A. Forouzan. *Data Communications and Networking*. McGraw-Hill, New York, NY, 5th edition, 2012. An in-depth textbook covering the principles of data communications and networking, including the OSI model.

[19] Robert Gebbers and Viacheslav I. Adamchuk. Precision agriculture and food security. *Science*, 327(5967):828–831, 2010.

[20] André Glória, Francisco Cercas, and Nuno Souto. Design and implementation of an iot gateway to create smart environments. *Procedia Computer Science*, 109:568–575, 2017. 8th International Conference on Ambient Systems, Networks and Technologies, ANT-2017 and the 7th International Conference on Sustainable Energy Information Technology, SEIT 2017, 16-19 May 2017, Madeira, Portugal.

[21] Government of India. Agricultural statistics at a glance 2020, 2020. Accessed: 2023-07-27.

[22] Ali Grami. Chapter 11 - communication networks. In Ali Grami, editor, *Introduction to Digital Communications*, pages 457–491. Academic Press, Boston, 2016.

[23] Mohinder S. Grewal, Angus P. Andrews, and Christopher G. Bartone. *Global Navigation Satellite Systems, Inertial Navigation, and Integration.* John Wiley Sons, 2020.

[24] Ajay Chandra V Gummalla and John O Limb. Wireless medium access control protocols. *IEEE Communications Surveys & Tutorials*, 3(2):2–15, 2000.

[25] Joaquín Gutiérrez, Juan Felipe Villa-Medina, Alejandra Nieto-Garibay, and Miguel Ángel Porta-Gándara. Automated irrigation system using a wireless sensor network and gprs module. *IEEE Transactions on Instrumentation and Measurement*, 63(1):166–176, 2014.

[26] Urs Hunkeler, Hong Linh Truong, and Andy Stanford-Clark. Mqtt-s—a publish/subscribe protocol for wireless sensor networks. In *2008 3rd International Conference on Communication Systems Software and Middleware and Workshops (COMSWARE'08)*, pages 791–798. IEEE, 2008.

[27] IJCMAS. The use of pesticides and fertilizers in india. *International Journal of Current Microbiology and Applied Sciences*, 8(5), 2019. Accessed: 2023-07-27.

[28] Indian Council of Agricultural Research. Green revolution, 2023. Accessed: 2023-07-27.

[29] Rolando Inglés, Mariusz Orlikowski, and Andrzej Napieralski. A c++ shared-memory ring-buffer framework for large-scale data acquisition systems. In *2017 MIXDES-24th International Conference" Mixed Design of Integrated Circuits and Systems*, pages 161–166. IEEE, 2017.

[30] Kumud Jha, Anurag Doshi, Pankaj Patel, and Manan Shah. A comprehensive review on automation in agriculture using artificial intelligence. *Artificial Intelligence in Agriculture*, 2:1–12, 2019.

[31] J. Benton Jones. *Plant Nutrition and Soil Fertility Manual.* CRC Press, 2012.

[32] Andreas Kamilaris, Andreas Kartakoullis, and Francesc X. Prenafeta-Boldú. A review on the practice of big data analysis in agriculture. *Computers and Electronics in Agriculture*, 143:23–37, 2017.

[33] Oratile Khutsoane, Bassey Isong, and Adnan M Abu-Mahfouz. Iot devices and applications based on lora/lorawan. In *IECON 2017-43rd Annual Conference of the IEEE Industrial Electronics Society*, pages 6107–6112. IEEE, 2017.

[34] Hicham Klaina, Imanol Picallo Guembe, Peio Lopez-Iturri, Miguel Ángel Campo-Bescós, Leyre Azpilicueta, Otman Aghzout, Ana Vazquez Alejos, and Francisco Falcone. Analysis of low power wide area network wireless technologies in smart agriculture for large-scale farm monitoring and tractor communications. *Measurement*, 187:110231, 2022.

[35] Thorsten Kramp, Rob Van Kranenburg, and Sebastian Lange. Introduction to the internet of things. *Enabling things to talk: Designing IoT solutions with the IoT architectural reference model*, pages 1–10, 2013.

[36] Zhiqiang Li, Ning Wang, and Jianping Chen. A wireless smart sensor network based on the zigbee technology for monitoring of an agricultural environment. *Journal of Sensors*, 2014:1–11, 2014.

[37] MathWorks. Thingspeak: An iot platform, 2023. Official website for ThingSpeak, an IoT analytics platform service.

[38] JM McKinion, SB Turner, JL Willers, JJ Read, JN Jenkins, and John McDade. Wireless technology and satellite internet access for high-speed whole farm connectivity in precision agriculture. *Agricultural Systems*, 81(3):201–212, 2004.

[39] Ministry of Finance, Government of India. Economic survey 2011-12, 2012. Accessed: 2023-07-27.

[40] Quan Nguyen. *Mastering Concurrency in Python: Create faster programs using concurrency, asynchronous, multithreading, and parallel programming.* Packt Publishing Ltd, 2018.

[41] Shashank M. Patil, Nishikant R. Kale, and Sandeep D. Lokhande. A review on internet of things for smart agriculture: Enabling techniques, applications, and scope. *International Journal of Electrical and Computer Engineering*, 7(6):3412–3422, 2017.

[42] David Pimentel. Environmental and economic costs of the application of pesticides primarily in the united states. *Environment, Development and Sustainability*, 7(2):229–252, 1995.

[43] Prabhu L. Pingali. Green revolution: Impacts, limits, and the path ahead. *Proceedings of the National Academy of Sciences*, 109(31):12302–12308, 2012.

[44] Nitin Rathore, Ramesh R. Naik, Sunil Gautam, Ravi Nahta, and Sidharth Jain. Smart farming based on iot-edge computing: Exploiting microservices' architecture for service decomposition. In Neha Sharma, Amol C. Goje, Amlan Chakrabarti, and Alfred M. Bruckstein, editors, *Data Management, Analytics and Innovation*, pages 425–437, Singapore, 2024. Springer Nature Singapore.

[45] Partha Pratim Ray. A survey on internet of things architectures. *Journal of King Saud University-Computer and Information Sciences*, 30(3):291–319, 2018.

[46] Luis Ruiz-Garcia, Leonardo Lunadei, Pilar Barreiro, and Juan Ignacio Robla. A review of wireless sensor technologies and applications in agriculture and food industry: State of the art and current trends. *Sensors*, 9(6):4728–4750, 2009.

[47] Olivier Bernard Andre Seller. Wireless communication method, 2016.

[48] Sam Siewert and John Pratt. *Real-Time Embedded Components and Systems with Linux and RTOS*. Mercury Learning and Information, 2015.

[49] MODBUS Application Protocol Specification. Modbus application protocol specification v1. 1b3. *Modbus_Application_Protocol_V1_1b3. pdf*, 2012.

[50] Espressif Systems. *ESP32 Technical Reference Manual*. Espressif Systems, 2020.

[51] Andrew S. Tanenbaum. *Computer Networks*. Prentice Hall, Upper Saddle River, NJ, 4th edition, 2003. A comprehensive text on computer networking, including detailed discussions on the OSI model and network protocols.

[52] Mostafa Tofighbakhsh and Bryan Lee Sullivan. Management of massively distributed internet of things (iot) gateways based on software-defined networking (sdn) via fly-by master drones, January 5 2021. US Patent 10,887,001.

[53] Bao Tran and Ha Tran. Iot-based farming and plant growth ecosystem, December 7 2021. US Patent 11,195,015.

[54] Eben Upton and Gareth Halfacree. *Raspberry Pi 4: The Complete Guide*. Raspberry Pi Foundation, 2020.

[55] George Vellidis, Mark Tucker, Carlton Perry, Charles Kvien, and Craig Bednarz. A real-time wireless smart sensor array for scheduling irrigation. *Computers and Electronics in Agriculture*, 61(1):44–50, 2008.

[56] C. N. Verdouw, J. Wolfert, A. J. M. Beulens, and A. Rialland. Virtualization of food supply chains with the internet of things. *Journal of Food Engineering*, 176:128–136, 2016.

[57] Ning Wang, Naiqian Zhang, and Maohua Wang. Wireless sensors in agriculture and food industry—recent development and future perspective. *Computers and Electronics in Agriculture*, 50(1):1–14, 2006.

[58] Sjaak Wolfert, Lan Ge, Cor Verdouw, and Marc-Jan Bogaardt. Big data in smart farming – a review. *Agricultural Systems*, 153:69–80, 2017.

[59] Wang Minghua Zhang, Na and Ning Wang. Precision agriculture—a worldwide overview. *Computers and Electronics in Agriculture*, 36(2-3):113–132, 2002.

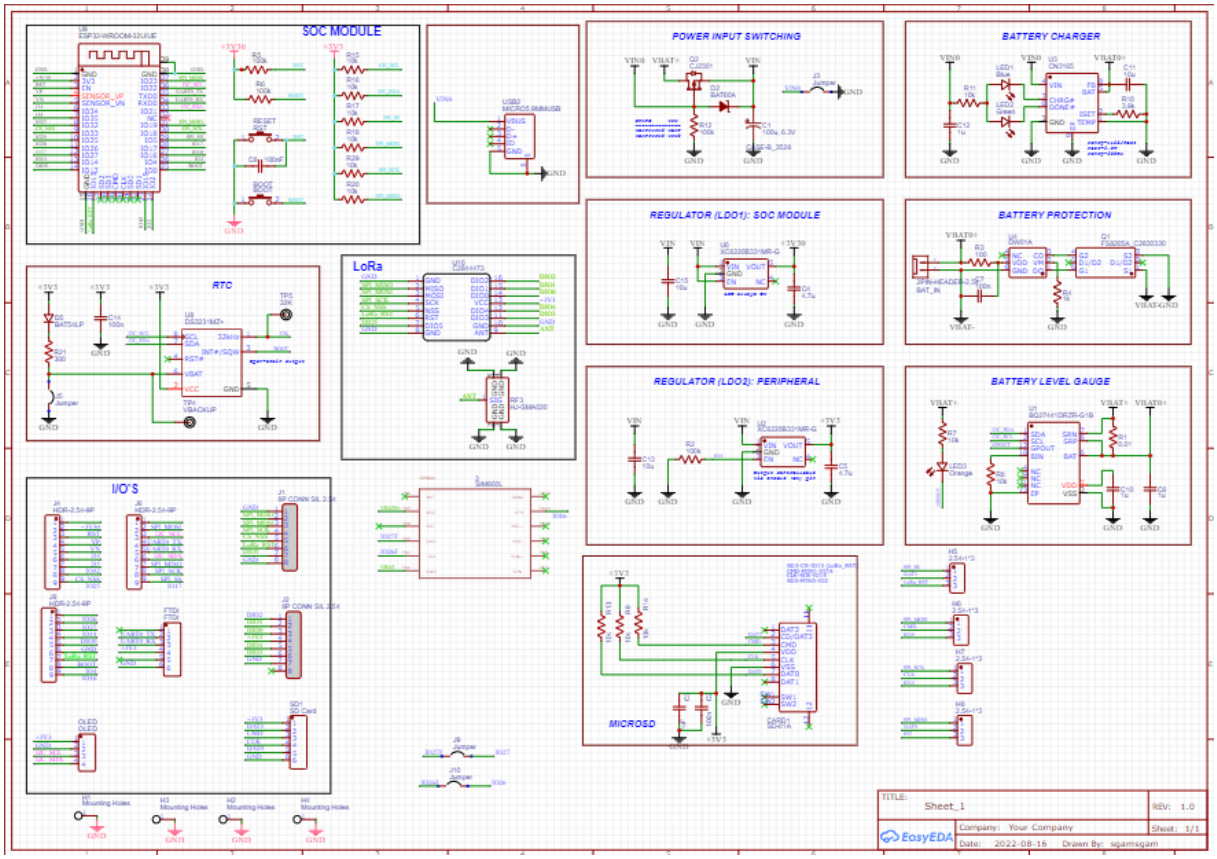

Figure A.1: Sheet 1 of the schematics of the node hardware module

A | Hardware Schematics of the Node Hardware

The hardware architecture of the IoT node for smart agriculture, which is based on the ESP32 WROOM32UE, has several key components to ensure robust, dependable, and adaptable performance in a variety of agricultural scenarios. This section provides a detailed description of the hardware components and features that are integral to the Internet of Things node. The node's PCB has been equipped with the following primary features:

- Power Input Switching;
- Battery Charging and Protection;
- Voltage Regulation;
- File Storage;
- LoRa Communication;
- Real-Time Clock (RTC);
- Connectivity Options - WiFi, BLE, GSM;
- Human-Machine Interface (HMI) Display - OLED;
- Programming;
- RS485 interfaces; and
- 4-20mA interfaces

The following figures show the schematic of the different hardware features of the node hardware.

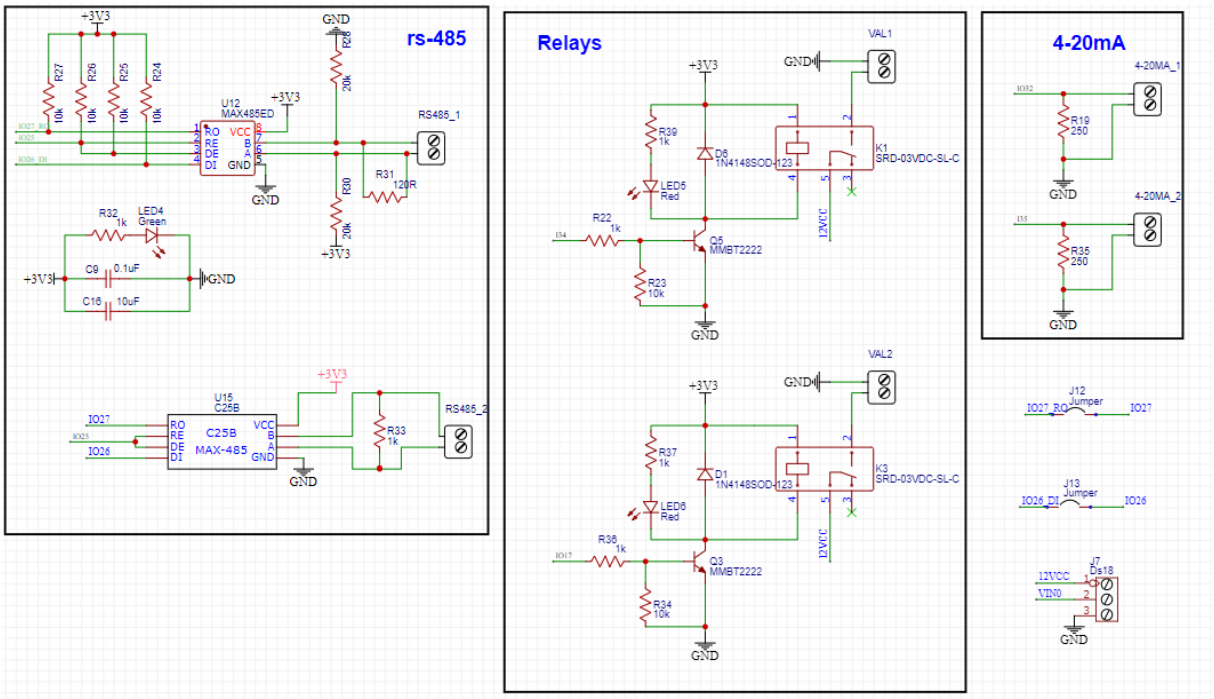

Figure A.2: Sheet 2 of the schematics of the node hardware module

B | Node Enclosure

This appendix is about the design and assembly of the node hardware enclosure. It is designed by the Mechanical Design and Fabrication (MDF) unit of CSIR-CSIO. The enclosure design has three major components: box, bracket and cover. They have been fabricated using Acrylonitrile Butadiene Styrene (ABS) material with the Fused Deposition Modeling (FDM) technique. The following subsections show the drawing of the enclosure.

B.1 | The Box

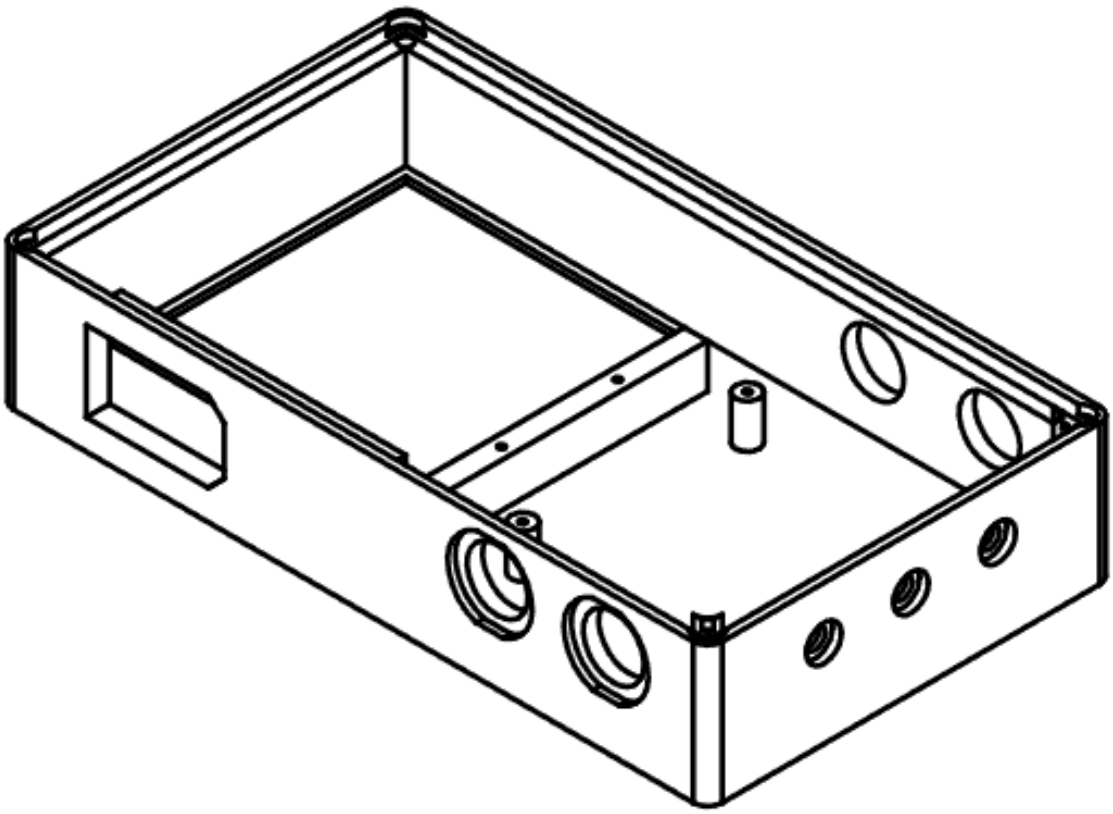

Figure B.1: Isometric View of the node enclosure box.

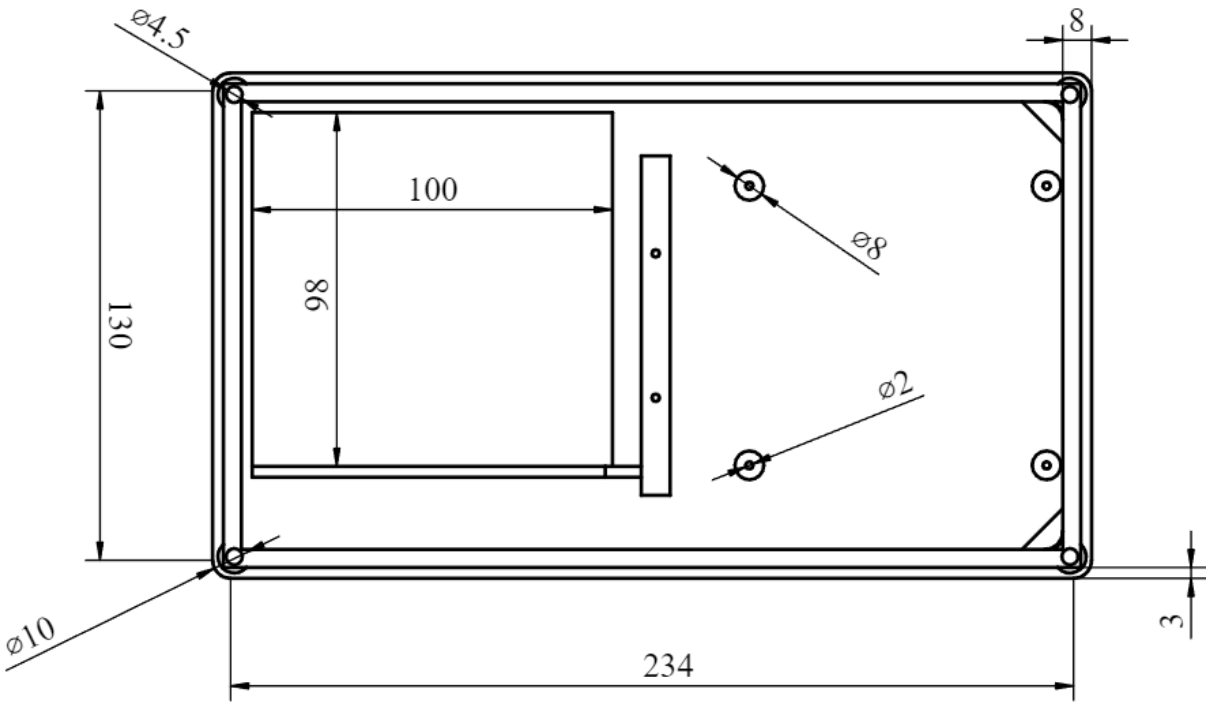

Figure B.2: Top view of the node enclosure box. All dimensions are in millimetres (mm)

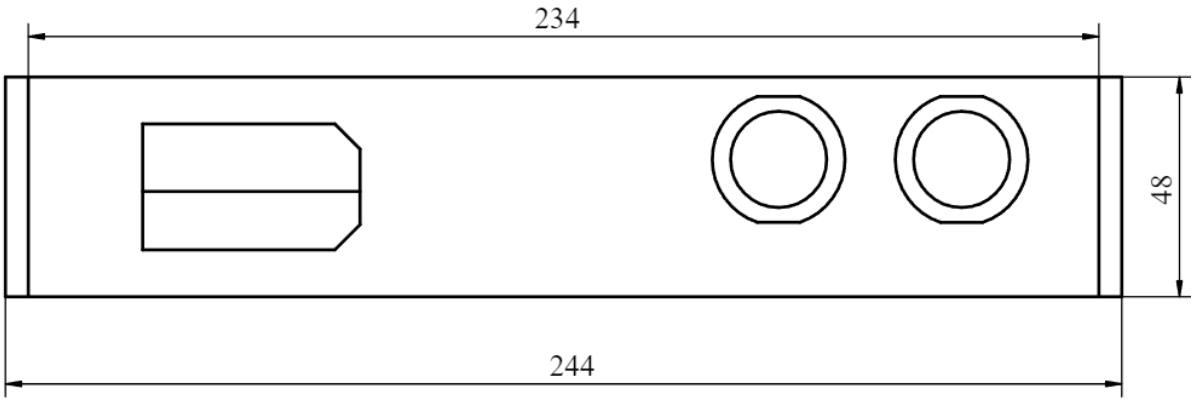

Figure B.3: Front view of the node enclosure box. All dimensions are in millimetres (mm)

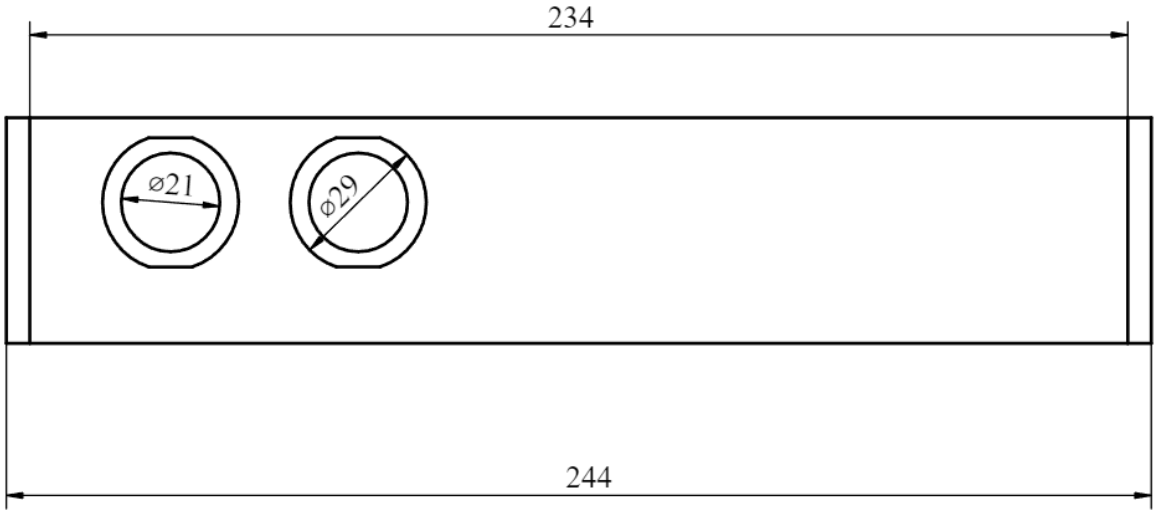

Figure B.4: Rear view of the node enclosure box. All dimensions are in millimetres (mm)

Figure B.5: Right side view of the node enclosure box. All dimensions are in millimetres (mm)

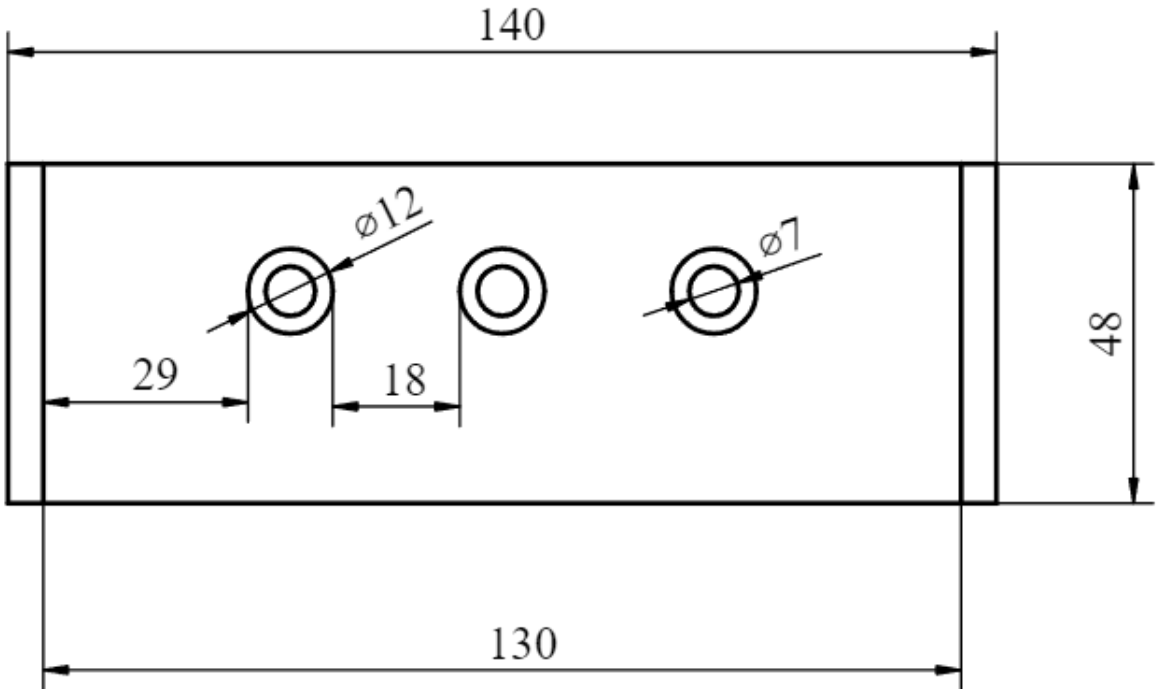

Figure B.6: Left side view of the node enclosure box. All dimensions are in millimetres (mm)

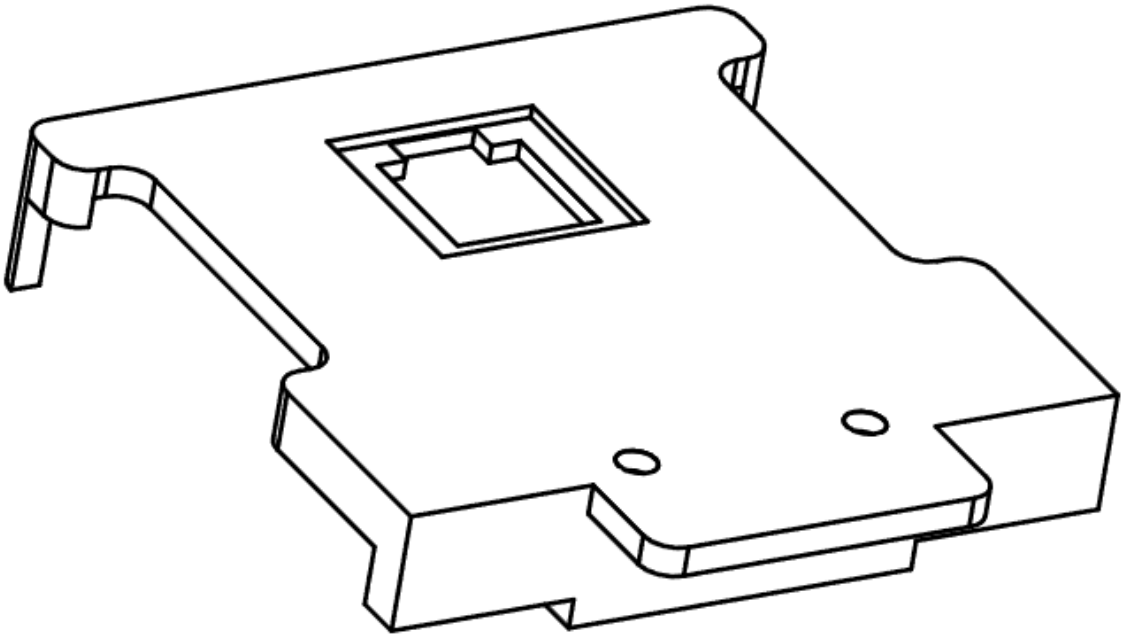

Figure B.7: Isometric view of the node bracket.)

B.2 | The Bracket

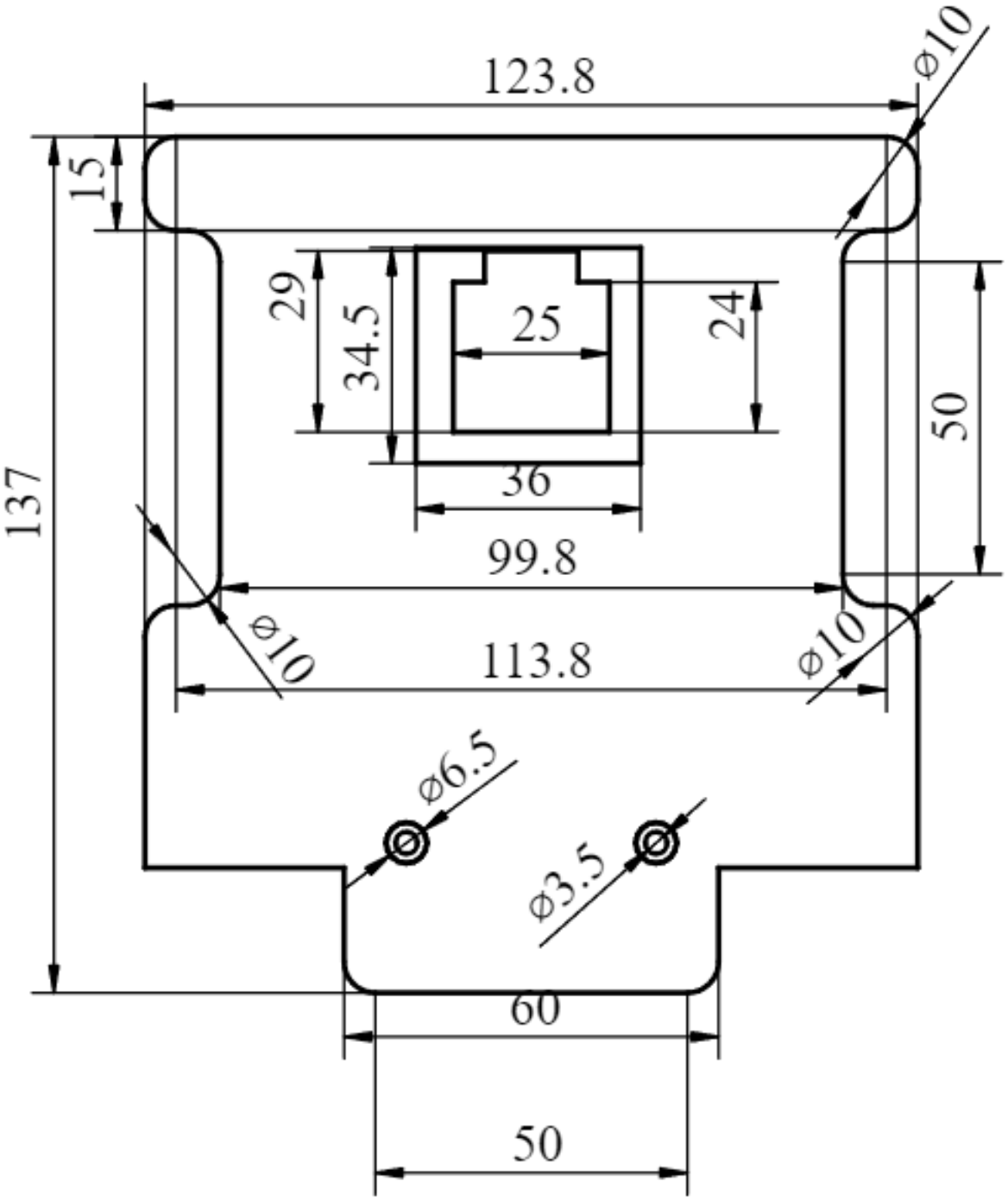

Figure B.8: Top view of the node bracket. All dimensions are in millimetres (mm)

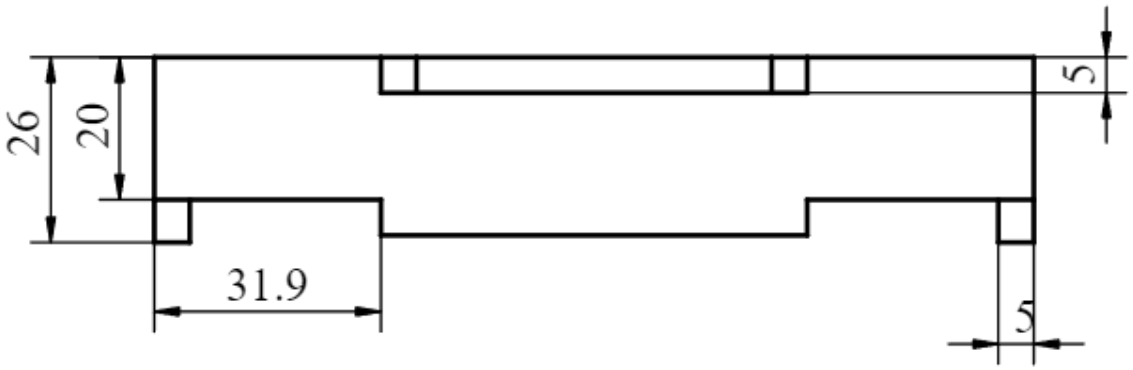

Figure B.9: Front view of the node bracket. All dimensions are in millimetres (mm)

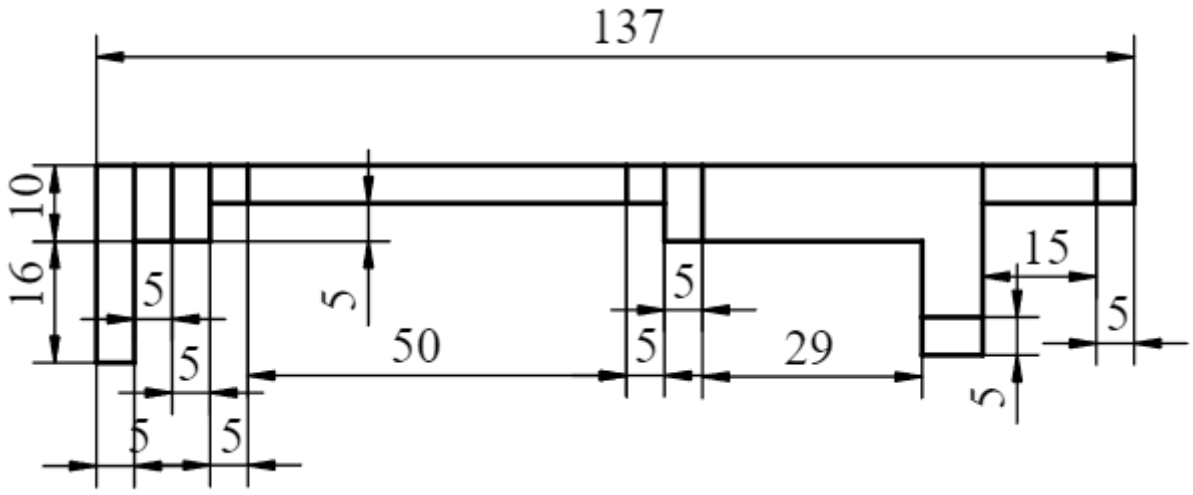

Figure B.10: Left side view of the node bracket. All dimensions are in millimetres (mm)

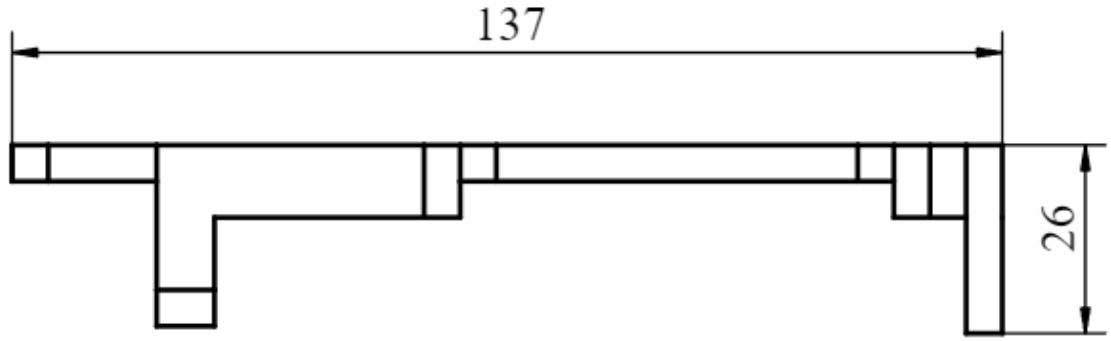

Figure B.11: Right side view of the node bracket. All dimensions are in millimetres (mm)

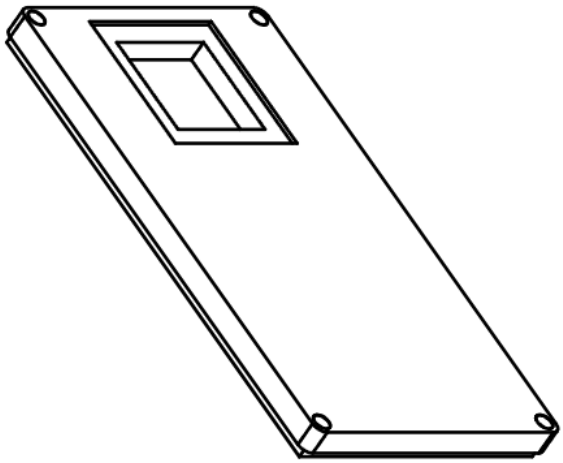

Figure B.12: Isometric View of the node cover box.

B.3 | The Cover

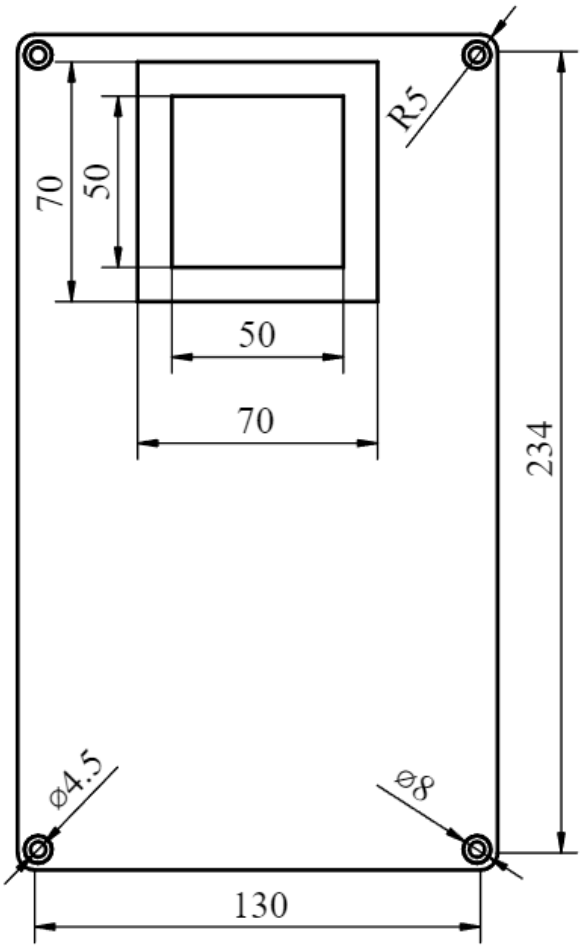

Figure B.13: Top view of the node cover. All dimensions are in millimetres (mm)

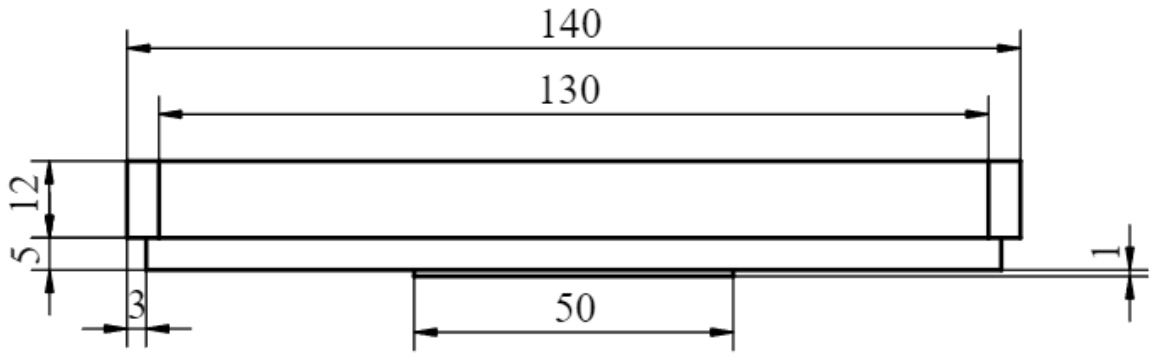

Figure B.14: Front view of the node cover. All dimensions are in millimetres (mm)

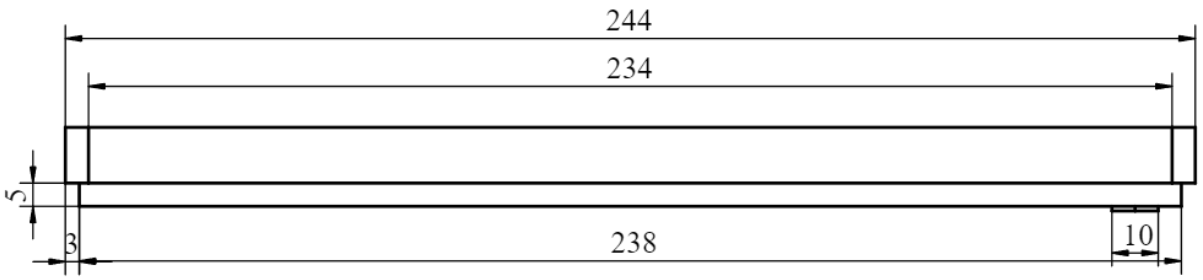

Figure B.15: Left side view of the node cover. All dimensions are in millimetres (mm)

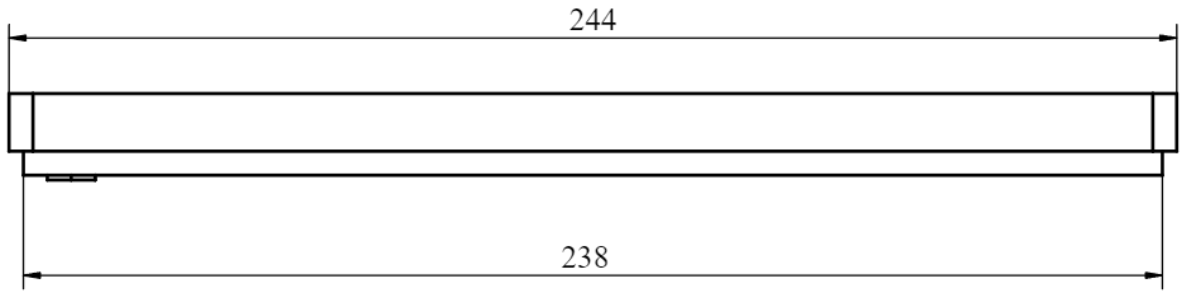

Figure B.16: Right side view of the node cover. All dimensions are in millimetres (mm)

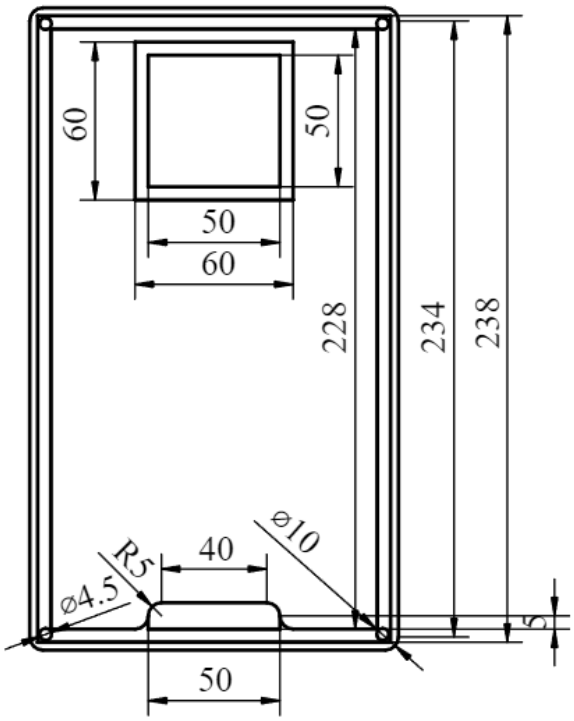

Figure B.17: Bottom view of the node bracket. All dimensions are in millimetres (mm)

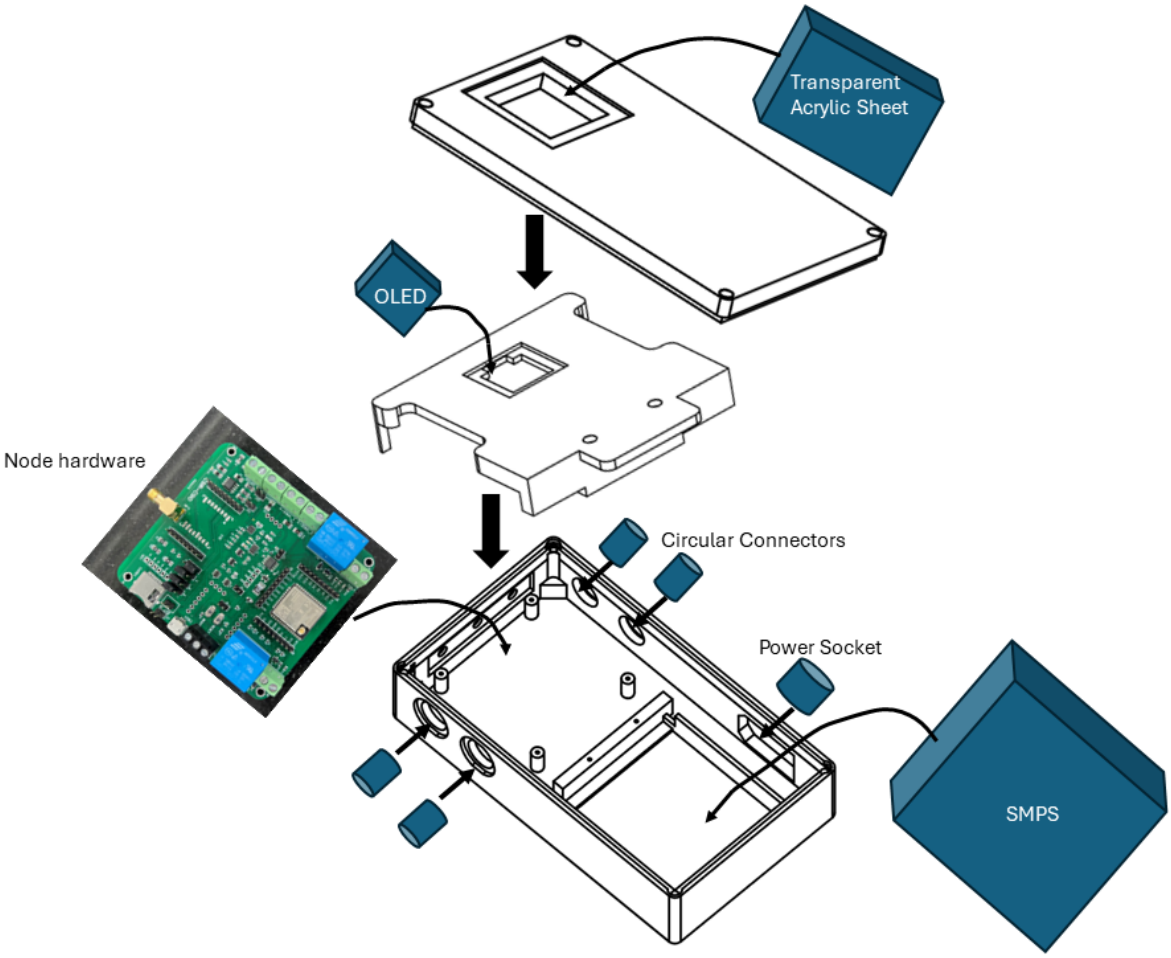

Figure B.18: Assembly process of the node hardware.

B.4 | Assembly

In the assembly process, first, the node PCB is affixed on the spacers with the LoRa antenna port protruding out of the box through its incision. Then, the SMPS is affixed at the demarcated location for it. At the incisions in the side, the power socket and 4-pin connectors are installed. Then, the bracket is fitted above the PCB. On the bracket, the OLED display and the GPS modules are installed. Then, the cover is put on the whole box-bracket installation. The GPS and WiFi antennas are connected to the SMA antenna outlets besides the LoRa antenna outlet. The assembly process is shown in the following figure.

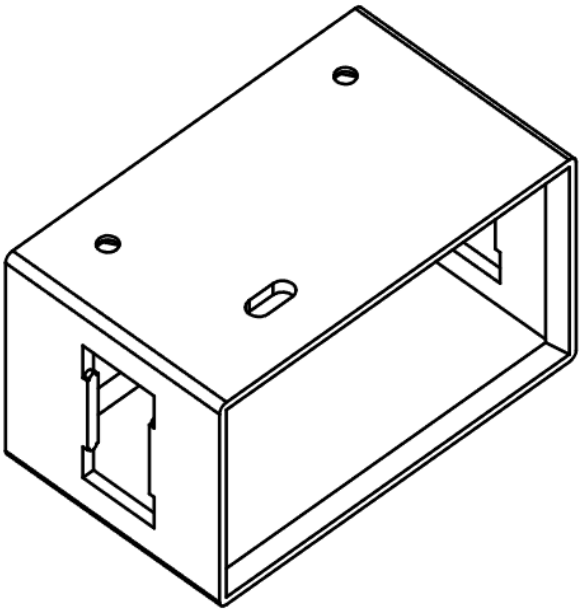

Figure C.1: Isometric view of the gateway enclosure box.

C | Gateway Enclosure

This appendix is about the design and assembly of the gateway hardware enclosure. It is designed by the Mechanical Design and Fabrication (MDF) unit of CSIR-CSIO. The enclosure design has four major components: box, two vents and a back cover. They have been fabricated using Acrylonitrile Butadiene Styrene (ABS) material with the Fused Deposition Modeling (FDM) technique. The following subsections show the drawing of the enclosure.

C.1 | The Box

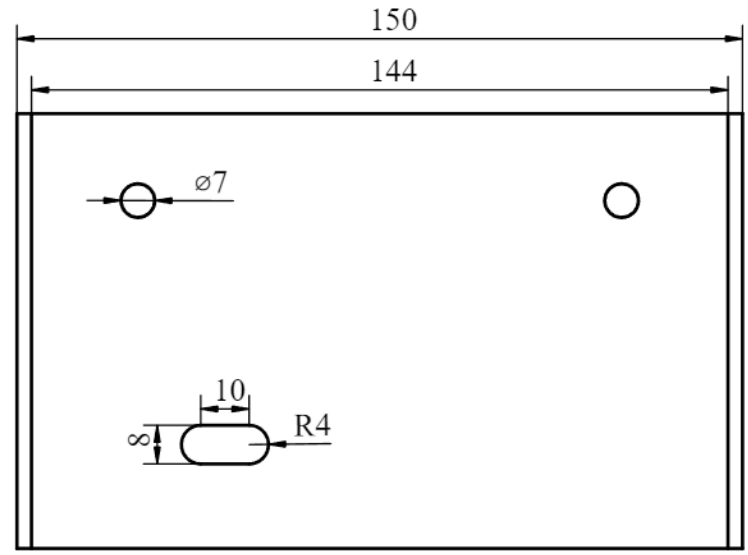

Figure C.2: Top view of the gateway enclosure box. All measurements are in millimetres (mm).

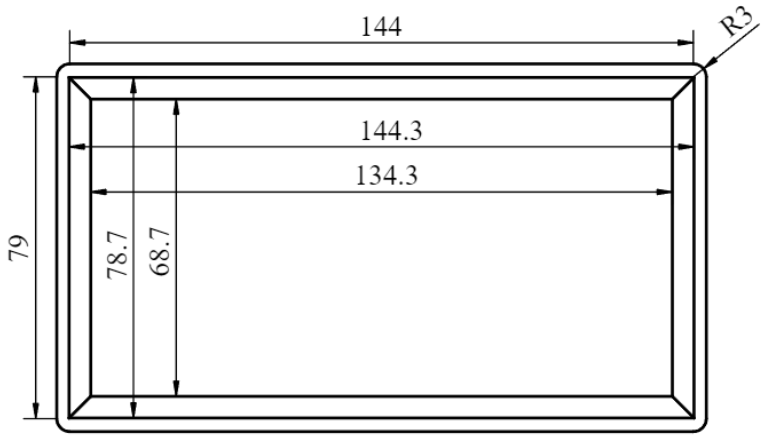

Figure C.3: Front view of the gateway enclosure box. All measurements are in millimetres (mm).

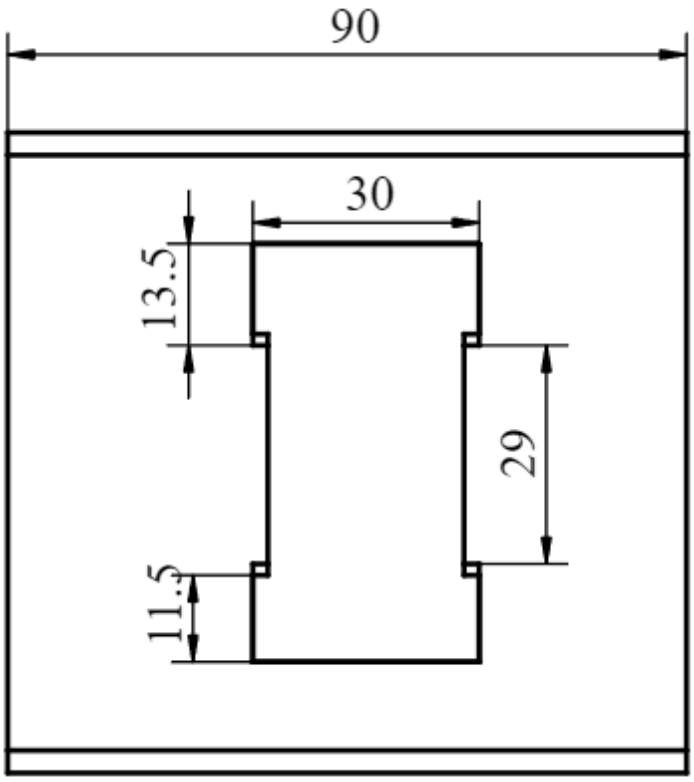

Figure C.4: Side view of the gateway enclosure box. All measurements are in millimetres (mm).

C.2 | The Vents

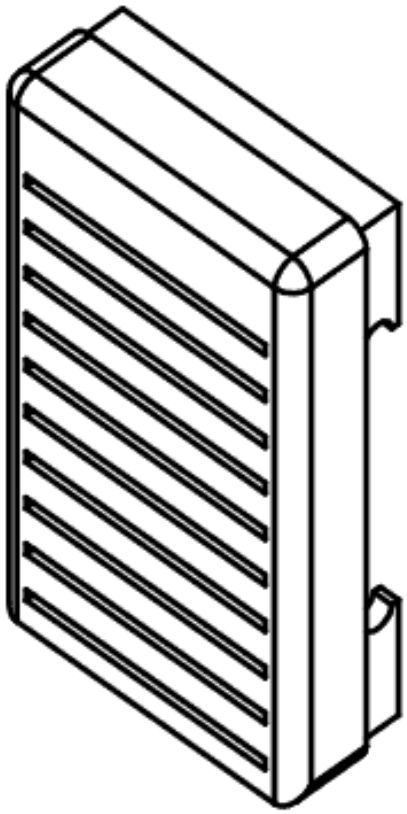

Figure C.5: Isometric view of the gateway enclosure vent.

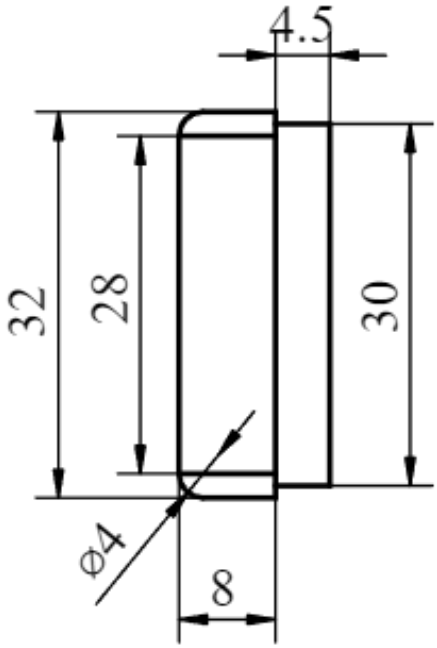

Figure C.6: Top view of the gateway enclosure vent. All measurements are in millimetres (mm).

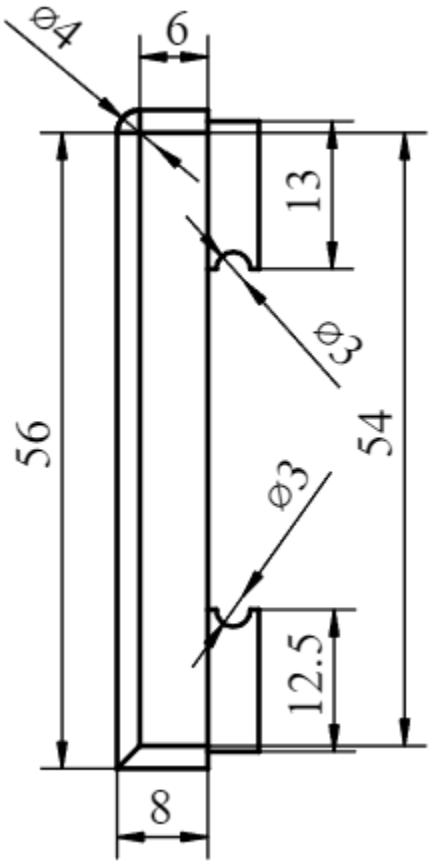

Figure C.7: Front view of the gateway enclosure vent. All measurements are in millimetres (mm).

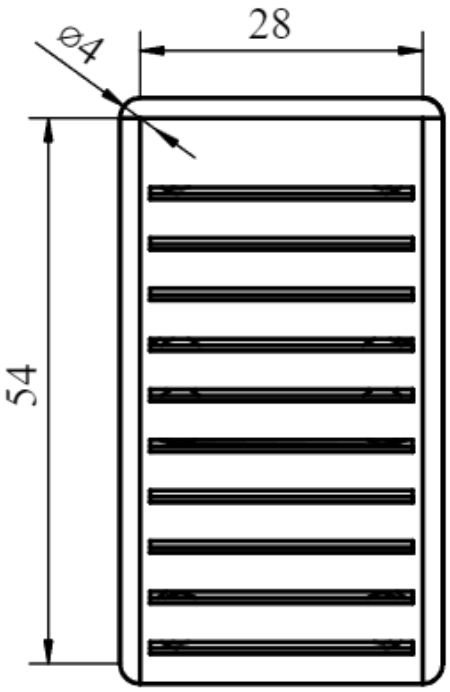

Figure C.8: Left side view of the gateway enclosure vent. All measurements are in millimetres (mm).

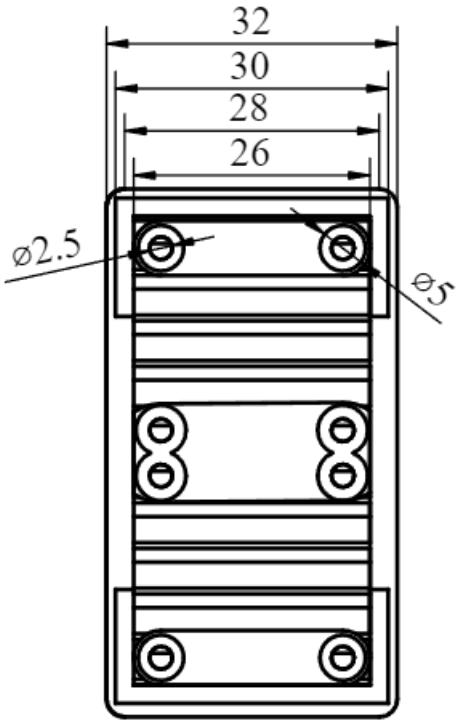

Figure C.9: Right side view of the gateway enclosure vent. All measurements are in millimetres (mm).

C.3 | The Back Cover

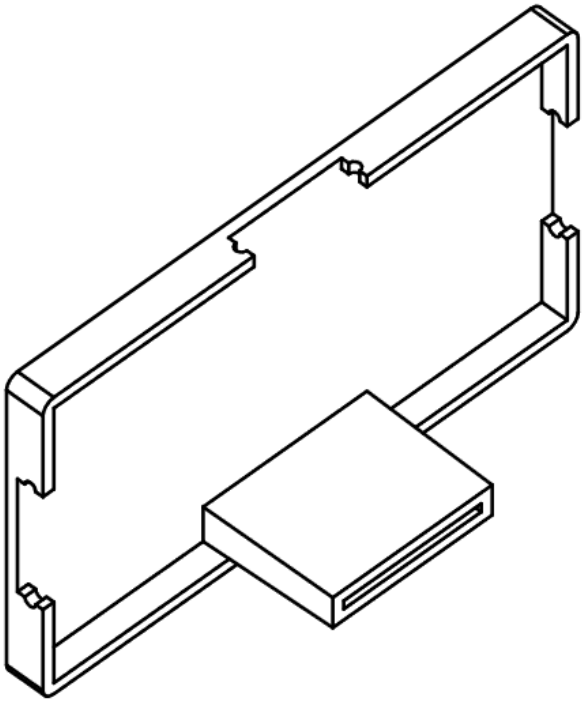

Figure C.10: Isometric view of the gateway enclosure cover.

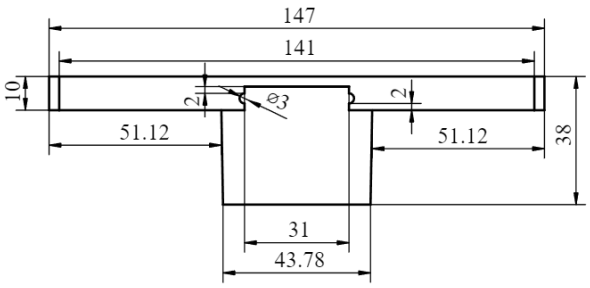

Figure C.11: Top view of the gateway enclosure cover. All measurements are in millimetres (mm).

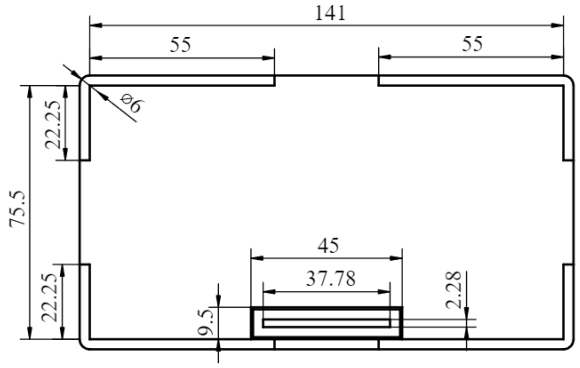

Figure C.12: Front view of the gateway enclosure cover. All measurements are in millimetres (mm).

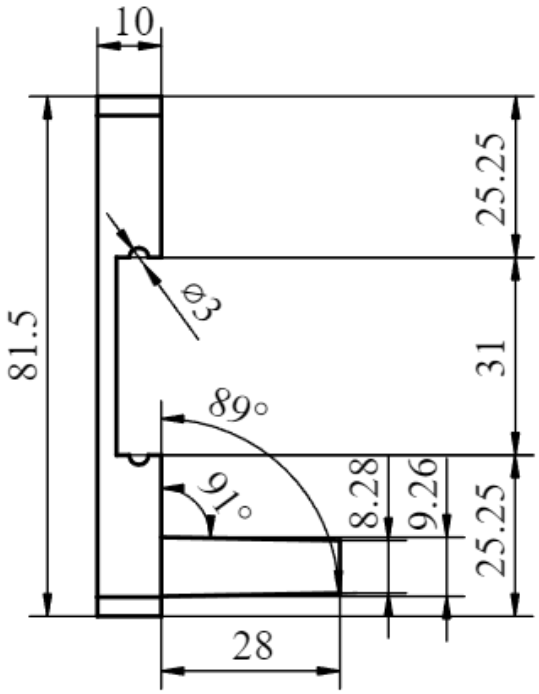

Figure C.13: Side view of the gateway enclosure cover. All measurements are in millimetres (mm).

C.4 | Assembly

In the assembly process, first, the LCD display connected to the Raspberry Pi is inserted into the box facing downwards. Then, a SIM7600X G-H hat is connected to the Raspberry Pi through the GPIO pins, also called pogo pins. For sturdiness, the SIM7600X G-H hat is fitted to the Raspberry Pi with the help of spacers. The UFL to SMA converters are connected to the SIM7600X G-H hat and the SMA ends are fixed to their incisions on the box. Then, a fan is attached to one of the vents. This vent is then attached to its space to the side of the Raspberry Pi where there are no ports. Then, the other vent is attached to the opposite side. Then, the back cover is installed. The assembly process is shown in the following figure.

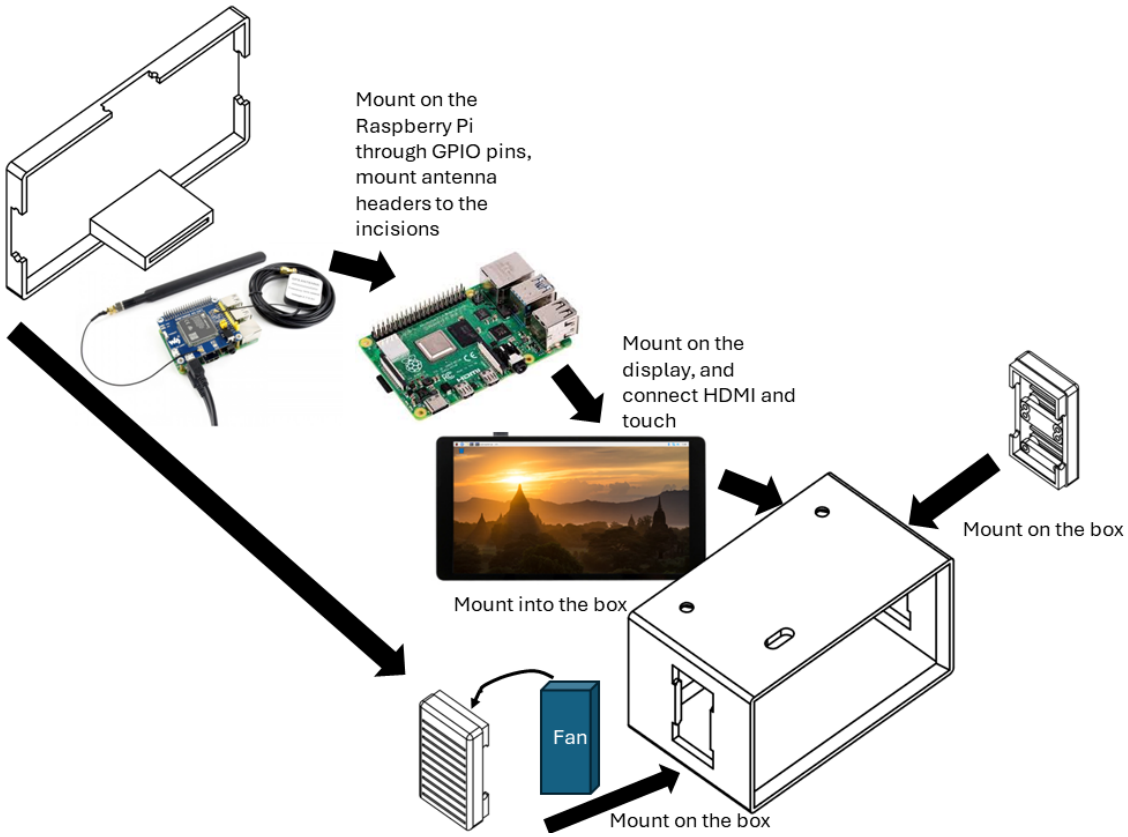

Figure C.14: Assembly process of the gateway hardware.

D | Miscellaneous Tasks

Apart from the development of devices for IoT network for Region Specific Smart Agriculture, I have completed the following tasks:

- **Documentation of Fluoride Sensing System:** I have coordinated with the team in our research group who have been developing the Fluoride Sensing System. Documentation included thorough literature review, electronics design, echanical chassis design, software development, and experiment documentation.
- **Chassis Design for Krishi-IoT device:** Apart from development of the software for the IoT devices, I also took the initiative to design the chassis for packaging both the node and the gateway devices.
- **Leaflets design and printing coordination:** I coordinated with the designer and printing services to have stickers and leaflets printed for some of the technologies developed by our research group. Unfortunately, it could not be taken further due to dispute with the service providers over unreasonable pricing provided after designing.
- **Development of indigeneous datalogger:** I initiated and spearheaded collaboration with Mindgrove Technologies Pvt. Ltd to co-develop an indigenous IoT datalogger based on Shakti microcontroller developed by IIT-Madras.
- **Procurement of Sensors:** I initiated contacts with different vendors to provide high quality soil, leaf and meteorological sensors for the purpose of this project. Before this, the features of different sensors and their sensing mechanisms have also been thoroughly vetted.
- **Coordination for renovation of laboratory space:** I have spearheaded the coordination efforts to procure furniture and designing the layout for the laboratory space meant for developing IoT devices for agriculture and water quality.

- **Coordination with Administration:** I have coordinated with the administration towards pushing our required purchasing requirements. This included timely submission of bills to the Stores and Purchases section, regularly taking updates from the Accounts section, and regularly addressing the queries raised by these sections for timely payments to the vendors for keeping a longstanding and trustworthy relationship to have access to high-quality materials useful for this project.
- **Coordination within lab group:** I have coordinated within the lab group for a number of tasks, which include creating inputs for the annual reports for CSIO, coordinating the purchases from different vendors and taking their status from the administration.