AGRICULTURAL AUTOMATED IRRIGATION SYSTEM

Unmesh Phaterpekar

Final Project Report
ECEN 5613 Embedded System Design
December 17, 2023

1 INTRODUCTION

The "Agricultural Automated Irrigation System" is a cutting-edge project that combines moisture and nutrient sensors to optimize plant growth. It uses moisture sensors to ensure plants receive the precise amount of water needed, thereby conserving water, and reducing waste. Simultaneously, nutrient sensors detect essential soil components like nitrogen, phosphorus, and potassium, informing if additional fertilization is needed. This system offers significant benefits, including enhanced plant growth, cost savings on water and nutrients, improved environmental sustainability through reduced resource usage, and data-driven insights for better crop management, ultimately leading to higher yields and quality in agricultural produce.

1.1 Overview

To give an overview, the main objective of the project is to develop automated irrigation systems capable of assessing soil moisture levels. These systems will identify whether the moisture in the soil is adequate for ideal plant growth or if additional watering is required. Furthermore, the project places considerable emphasis on detecting the levels of key nutrients such as nitrogen, phosphorus, and potassium in the soil.

Figure 1.1 given is an overview of how I have interfaced the sensor and other things such as the relay, water pump and LEDs. Also, how I have interfaced the soil NPK sensor with Arduino UNO using the Modbus Protocol.

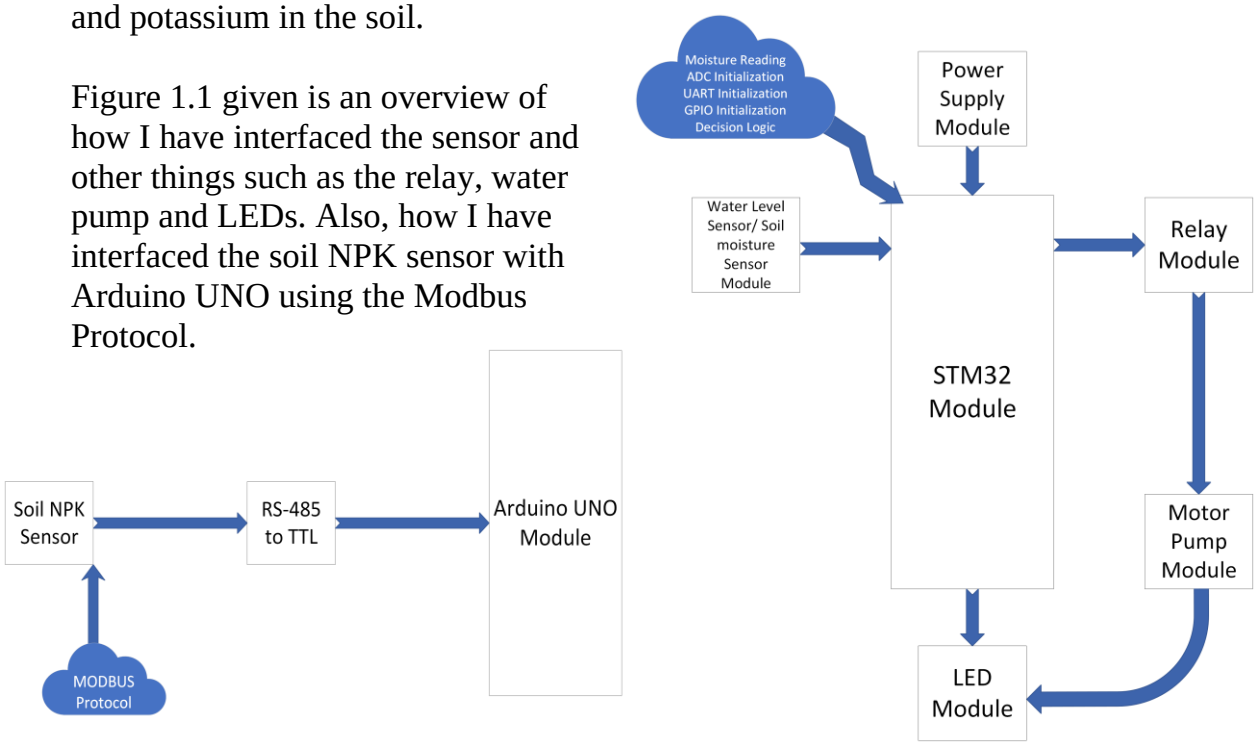


Figure 1.1

2 TECHNICAL DESCRIPTION

This part of the documentation outlines its technical framework, emphasizing hardware implementation and sensor interfacing. It begins by detailing the use of Analog-to-Digital Conversion and the RS485 interface, integrated with the Modbus Protocol. This provides insights into the sensors' communication with the main system. Additionally, the document covers the board design, highlighting the integration of various hardware components with the STM32 microcontroller and the Arduino platform. This offers a comprehensive view of the hardware architecture and its connections. For clarity, the hardware is divided into two sections. The first, 'New Hardware #1 Description', explains the water level sensor's functionality and role in the system. The second, 'New Hardware #2 Description', focuses on the soil NPK sensor, discussing its importance in monitoring soil health and data acquisition. The firmware design section addresses the programming aspects, emphasizing code development for microcontroller communication. This includes the software-hardware interface, focusing on sensor data processing and management logic. Finally, the software design segment is dedicated to the user interface and data presentation. It outlines methods for displaying sensor data on a terminal window, making it accessible and interpretable for users. This is vital for effective monitoring and decision-making, translating raw sensor data into usable information.

Figure 2 given below depicts the all the hardware connections I have made for interfacing the sensors.

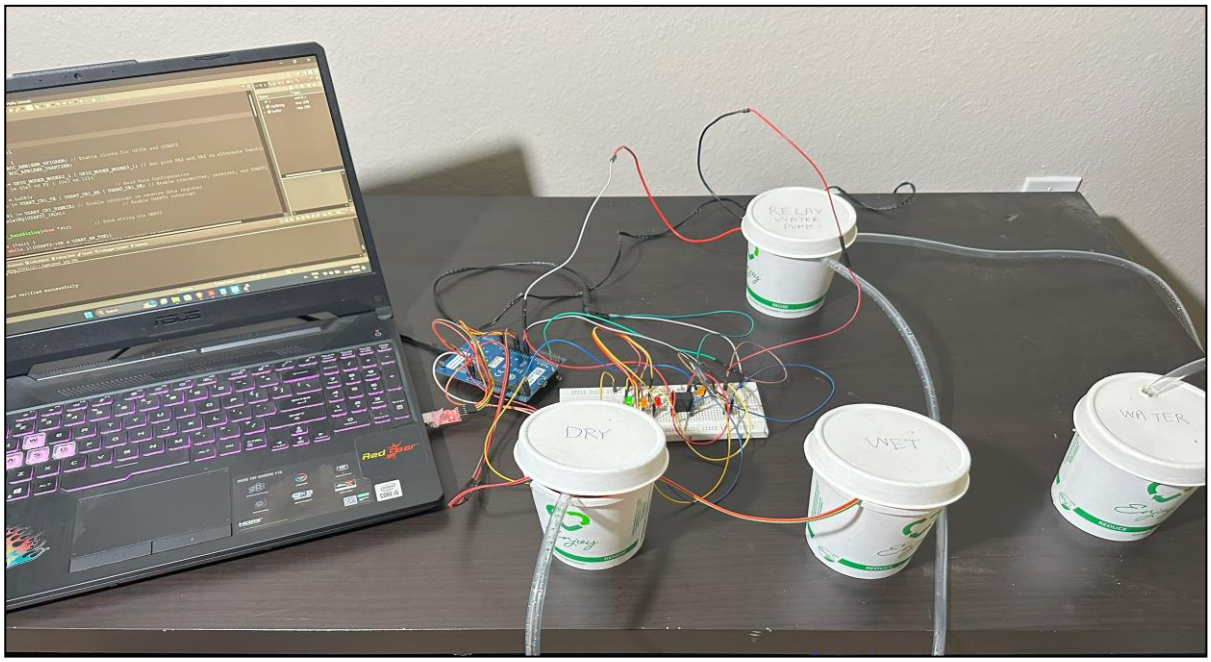


Figure 2

2.1 Board Design

The project incorporates a sophisticated array of hardware components, meticulously chosen to optimize its functionality. Central to the system's operation are two pivotal microcontroller units: the STM32F411E-Discovery board and the Arduino UNO. These units collectively serve as the system's computational and control hub, orchestrating the various sensors and actuators in a harmonious manner. A key element in this setup is the integration of water level sensors. These sensors play a critical role in monitoring soil moisture levels, ensuring that the water content within the soil is maintained at an optimal level for plant growth. This monitoring is crucial for both conserving water and promoting healthy plant development.

To facilitate the automated watering of plants, a relay is employed to control a water pump. This pump is designed to deliver precise amounts of water, calibrated to meet the specific hydration needs of the plants, thus preventing both under and over-watering.

Another innovative feature of this system is the inclusion of a soil NPK sensor. This sensor interfaces directly with the Arduino, providing vital data on the concentration of nitrogen, phosphorus, and potassium in the soil. This information is instrumental in assessing the nutrient richness of the soil, enabling informed decisions on whether additional fertilization is necessary to achieve optimal plant growth.

Additionally, the system incorporates a small pipe that efficiently channels water from the source to the targeted soil areas. This ensures a direct and controlled water delivery mechanism, further enhancing the efficiency of the irrigation process.

Lastly, the design includes an NPN transistor, a crucial component that amplifies the current output from the STM32F411E. This amplification is necessary to generate sufficient current for driving the water pump through the relay. The transistor's role is pivotal in ensuring that the electrical requirements of the pump are adequately met, thereby ensuring smooth and reliable operation of the watering system.

Overall, this carefully engineered setup represents a harmonious integration of electronic components and sensors, all working in tandem to create a highly efficient and automated plant watering system.

Figure 2.1 given below gives a general idea of the components I have used.

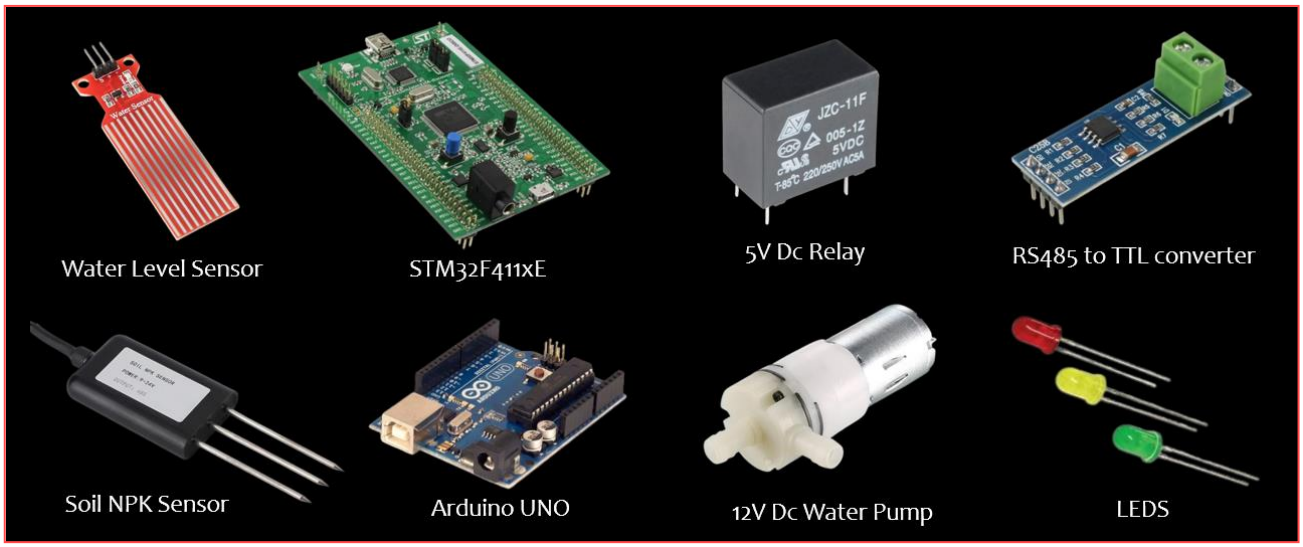


Figure 2.1

2.2 New Hardware #1 – Water Level Sensor

The water level sensor module, a vital component recently integrated with the STM32F411E microcontroller, plays a crucial role in measuring liquid volumes. This sensor, often referred to as a fluid or liquid level sensor, is engineered to accurately gauge the water content from top to bottom in various containers or environments. Commonly encountered in everyday technology, water level sensors are integral in systems such as vehicle fuel gauges, where they monitor the fuel level using similar principles.

Designed primarily for commercial and industrial applications, these sensors are also accessible for educational, and hobbyist uses. The module in question is a compact version of a conductivity liquid level sensor. It operates primarily on the principle of variable resistance to ascertain the water level within a container. Additionally, it possesses the capability to detect water droplets, making it a versatile tool for measuring precipitation or atmospheric moisture levels. This sensor is compatible with a range of popular microcontrollers and single-board computers, including STM32, Raspberry Pi, and Arduino Uno. Its construction features ten exposed copper traces, divided equally between sense traces, which detect the liquid level, and power traces, which facilitate the sensor's operation. Some variants of this sensor module are equipped with a power LED, which serves as an indicator of the module's active state.

The module's design is notably straightforward, encompassing three primary pins as shown in Figure 2.2 from “Digikey.com”:

Figure 2.2

1. Signal (S): This analog pin transmits signals to the microcontroller or single-board computer, conveying information about the fluid level.
2. VCC/5V (+): Serving as the power input pin, it is compatible with a voltage range of 3.3V to 5V.
3. GND (-): This pin establishes the ground connection.

These pins are essential for successfully interfacing the water level sensor with the STM32 microcontroller, ensuring reliable and precise fluid level measurements in various applications.

2.3 New Hardware #2 – Soil NPK Sensor

Additionally, I've successfully integrated a soil NPK sensor with an Arduino UNO, a crucial step in understanding and enhancing plant nutrition. Plants, akin to humans, need sustenance to thrive. The trio of key nutrients - Nitrogen (N), Phosphorus (P), and Potassium (K), collectively known as NPK - are vital for their robust growth and yield. Insufficient levels of these nutrients in garden soil can hinder plants from reaching their full growth potential. Monitoring the soil's NPK content is therefore critical to determine the necessary nutrient supplementation for optimal crop fertility.

Employing an Arduino-connected NPK Soil Sensor allows for the efficient assessment of soil nutrient levels. This sensor excels in detecting nitrogen, phosphorus, and potassium in the soil, playing a pivotal role in evaluating soil fertility. Its durability, resistant to long-term electrolysis and corrosion, and its waterproof design through vacuum potting, enable it to be permanently embedded in soil. The sensor finds its application in various domains such as precision agriculture, forestry, soil science, geological exploration, and plant cultivation.

The JXCT Soil NPK sensor stands out for its affordability, quick response, commendable accuracy, and portability, offering real-time insights into soil NPK content for advanced agricultural practices. This sensor specifically measures soil levels of nitrogen, phosphorus, and potassium, contributing to a comprehensive understanding of soil fertility. It operates efficiently within a voltage range of 5-30V and is power-efficient. Its datasheet reveals its capability to measure NPK with a precision of up to 1 mg/kg (mg/l).

As shown in Figure 2.3 taken from “Amazon.com”, the sensor is equipped with a durable stainless-steel probe, resistant to rust, electrolysis, and salt-alkali damage, making it suitable for various soil types including alkaline, acidic, substrate, seedling bed, and coconut bran soils. To prevent moisture ingress, the probe is securely sealed with high-density epoxy resin. A standout feature of the sensor is its IP68 rating, ensuring protection against dust and moisture, thereby promising longevity, and reliable performance.

Figure 2.3

For effective long-distance usage, the sensor incorporates an RS485 communication interface and supports the standard Modbus-RTU communication protocol, enhancing its versatility in various agricultural settings.

2.4 New Hardware #3 – Water pump

I'm utilizing a 12V DC water pump, primarily designed to transport water from its source to my plant pots. The evolution of home appliances and instrumentation industries has led to a surge in the popularity of compact, oil-free, and eco-friendly DC pumps. Predominantly, these mini water pumps are powered by DC, featuring an integrated design where the motor and pump cavity are combined and operated by a DC power supply. These are often referred to as DC miniature water pumps.

DC pumps come in various types, differentiated primarily by the rated DC voltage that powers the motor. One such example is the 12V DC motor-driven electric water pump as shown in Figure 2.4 taken from “Amazon.com”. Versatile in application, it can be used with a switching DC stabilized power supply for voltage transformation or connected directly to a car battery for convenience. These pumps are extensively employed in multiple fields, including water treatment, environmental protection, medical applications, industrial control, and scientific research. Their uses range from water circulation and cooling to lifting, transferring, pressurizing, spraying, and other similar purposes.

Figure 2.4

2.5 New Hardware #4 – Relay

Figure 2.5

The primary role of a relay is to generate sufficient force to operate a water pump, enabling it to efficiently transfer water to and from designated locations as needed. It allows for the control of high-current devices using a low-current trigger. The operating voltage for triggering this relay ranges between 0 to 5V. The structure of the 5V relay includes a 5-pin layout, each with specific functions, as outlined below and illustrated in Figure 2.5.

Pin1: This activates the relay. One end is usually connected to 5V, while the other end links to the ground. Pin2: Also used for relay activation. Pin3 (Common (COM)): Connects to the main terminal of the load for activation. Pin4 (Normally Closed (NC)): One of the load terminals can be connected here. When connected, the load remains ON before activation of the switch. Pin5 (Normally Open (NO)): Connecting the load's second terminal here means the load remains OFF until the switch is activated.

The relay module, particularly the single-channel board type, is designed to handle high voltage and current loads, such as solenoid valves, motors, AC loads, and lamps. It's primarily engineered for interfacing with various microcontrollers like STM32, PIC, Arduino, etc.

Normally Open (NO): This pin stays open until a signal is sent to the relay module's signal pin. Upon receiving a signal, the common contact pin breaks its connection with the NC pin and establishes a connection with the NO pin.

Common Contact: This pin facilitates connection with the load that needs to be switched using the module.

Normally Closed (NC): This pin forms a closed circuit with the COM pin. This connection is interrupted when the relay is activated by sending a high/low signal to the signal pin from a microcontroller.

Signal Pin: This pin controls the relay, operating in either active low or active high modes. In the active low mode, the relay is activated by a low signal, while in the active high mode, it activates with a high signal.

5V VCC: This pin requires a 5V DC power supply.
Ground: This pin connects to the GND terminal of the power source.

2.6 New Hardware #4 – MAX RS485 Transceiver

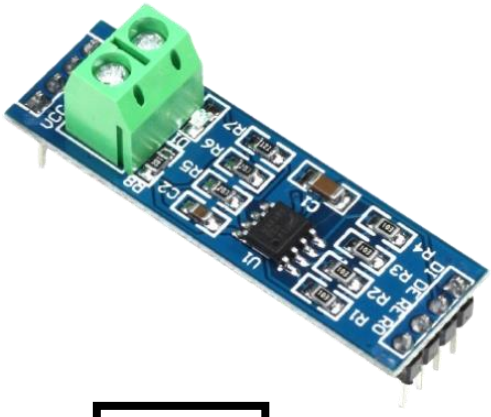


Figure 2.6

As shown in Figure 2.6, the transceiver module is designed around the Maxim MAX485 IC, offering a robust solution for serial communication over extended distances, up to 4000 feet (approximately 1200 meters). This module is adept at facilitating half-duplex serial data transmission, supporting data rates up to 2.5 Mbps. It's well-suited for a variety of microcontrollers, including popular ones like STM32, Arduino and PIC, thanks to its compatibility with a standard 5V power supply and 5V logic levels.

One of the key features of this module is its simple control mechanism for dictating the direction of communication, achievable through a single digital pin. This flexibility extends to the ability to connect multiple modules together, enabling communication between two or more devices. For scenarios requiring even longer distances, a pair of these modules can be configured as a repeater. Each unit is comprised of a driver and a receiver. The drivers boast short-circuit current limitation and are safeguarded by thermal shutdown circuitry, which triggers a high-impedance state in the driver outputs to prevent power over-dissipation. On the receiver side, the input includes a fail-safe feature, ensuring a logic-high output if the input is left open.

The module's pinout is as follows:

VCC: 5V Power Supply
A: Non-inverting Receiver Input and Non-inverting Driver Output
B: Inverting Receiver Input and Inverting Driver Output
GND: Ground (0V)
RO: Receiver Output (connects to the Rx pin of a microcontroller)
RE: Receiver Output Enable (activates when low)
DE: Driver Output Enable (activates when high)
DI: Driver Input (connects to the Tx pin of a microcontroller)

2.7 Firmware Design

Designed a detailed firmware for interfacing a water level sensor with the STM32F411E microcontroller, which involved ADC (Analog-to-Digital Converter), NPN 2222A transistor, 5V relay, LEDs for soil moisture detection, and a diode for back EMF protection. The process encompassed several steps. Here's a retrospective outline of the firmware design:

Initialization Phase was the microcontroller Setup which involved the STM32F411E microcontroller. Then I configured the clock system for the desired operation frequency.

2.7.1 Analog – to – Digital Conversion

The STM32F411E-Discovery has a 12-bit ADC. It has up to 19 multiplexed channels allowing it to measure signals from 16 external sources, two internal sources, and the VBAT channel. The A/D conversion of the channels can be performed in single, continuous, scan or discontinuous mode. The result of the ADC is stored into a left or right-aligned 16-bit data register. The watchdog feature allows the application to detect if the input voltage goes beyond the user-defined, higher, or lower thresholds.

As shown in Timing diagram 2.7.1.1 taken from the “STM32F411e datasheet”, the ADC needs a stabilization time of tSTAB before it starts converting accurately. After the start of the ADC conversion and after 15 clock cycles, the EOC flag is set, and the 16-bit ADC data register contains the result of the conversion.

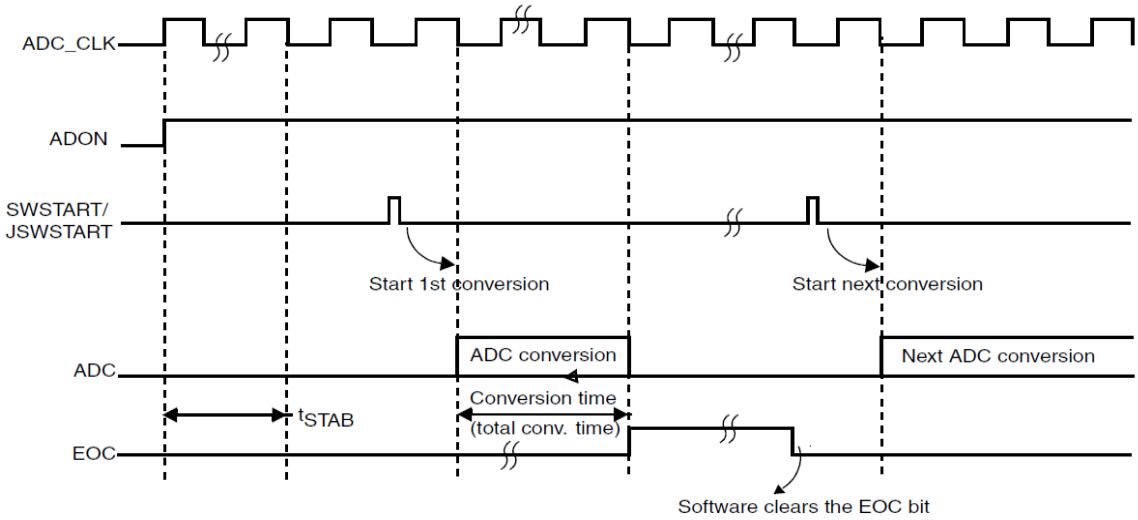


Figure 2.7.1.1

For reading values I am using the single conversion mode.

In this mode, the Analog-to-Digital Converter (ADC) performs a single conversion process. This process is initiated in two ways: for a regular channel, it's triggered by activating the ADON bit in the ADC_CR2 register; for either a regular or injected channel, it can be started through an external trigger. This occurs while the CONT bit is maintained at 0. After the designated channel's conversion is completed:

For a regular channel conversion:

- ➔ The 16-bit ADC_DR register stores the conversion result.
- ➔ The End Of Conversion (EOC) indicator becomes active.
- ➔ An interrupt is triggered if the EOCIE bit is engaged.
- ➔

For an injected channel conversion:

- ➔ The conversion result is stored in the 16-bit ADC_DRJ1 register.
- ➔ The End Of Conversion for Injected channels (JEOC) indicator is activated.
- ➔ An interrupt occurs if the JEOCIE bit is enabled.

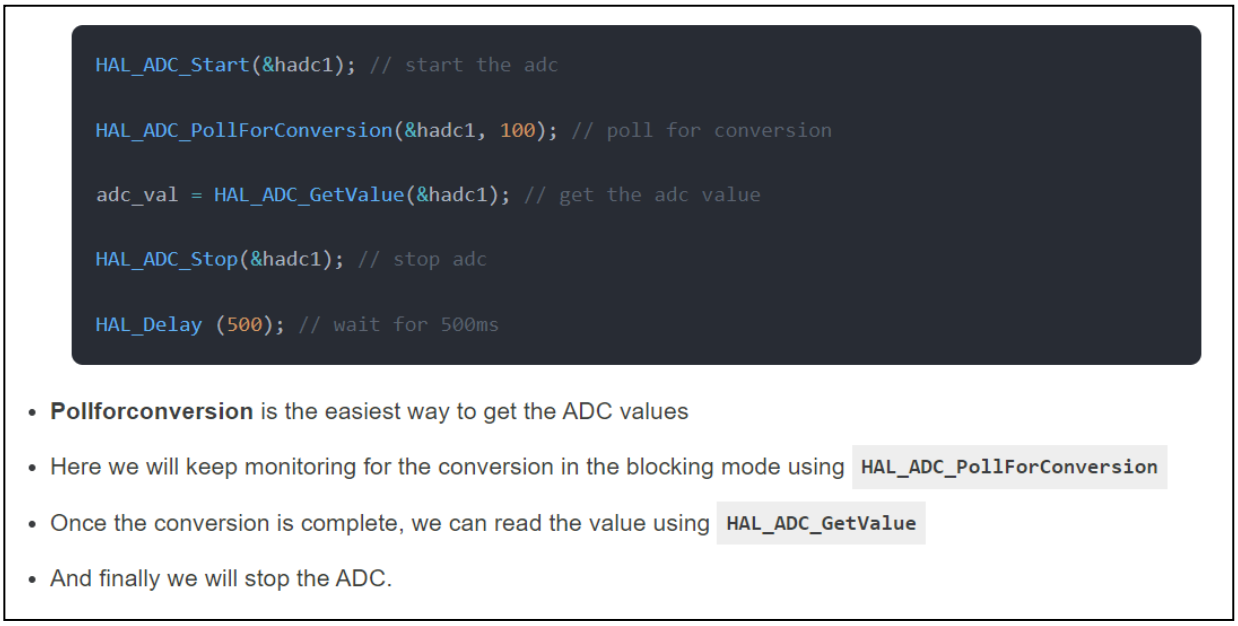
Subsequently, the ADC operation is halted.

I have implemented the following steps for ADC configuration.

1) Initialized the ADC by setting up the ADC clock, enabled the ADC, and set the ADC resolution (e.g. 12-bit).
2) Enabled the ADC's analog input channel ADC_Channel_0.
3) Set the ADC's reference voltage.
4) Set the ADC's gain ADC_Gain_6.
5) Start the ADC conversion - ADC_StartConversion().
6) Waiting for the ADC conversion to complete while (ADC_GetFlagStatus(ADC_FLAG_EOC) == RESET));
7) Read the ADC conversion result (e.g. ADC_Read()).
8) Calculated the water level based on the ADC conversion result and a calibration factor.
9) Stored the water level reading in a buffer or send it over a communication interface like UART.
10) Repeated the steps 5-9 at a desired interval to continuously monitor the water level.

There are different ways to read the STM32 ADC, but I am using the polling method because it's the easiest way to code to perform an analog to digital conversion using the ADC on an analog input channel. However, it's not an efficient way in all cases as it's considered to be a blocking way of using the ADC. As in this way, we start the A/D conversion and wait for the ADC until it completes the conversion so the CPU can resume processing the main code.

Given below in Figure 2.7.1.2 from "controllerstech.com" is a snippet of basic overview of how to implement the ADC for polling method using the HAL library in STM32 Cube IDE.



```
HAL_ADC_Start(&hadc1); // start the adc

HAL_ADC_PollForConversion(&hadc1, 100); // poll for conversion

adc_val = HAL_ADC_GetValue(&hadc1); // get the adc value

HAL_ADC_Stop(&hadc1); // stop adc

HAL_Delay (500); // wait for 500ms
```

- **Pollforconversion** is the easiest way to get the ADC values
- Here we will keep monitoring for the conversion in the blocking mode using `HAL_ADC_PollForConversion`
- Once the conversion is complete, we can read the value using `HAL_ADC_GetValue`
- And finally we will stop the ADC.

Figure 2.7.1.2

Before moving on directly to the water level sensor, I first tried to implement the ADC with the on-board temperature sensor.

The STM32F411e has an onboard temperature sensor whose result is a 12-bit value that represents the temperature in degrees Celsius. The temperature sensor has a range of -40°C to 85°C. The ADC result is a digital value that represents the temperature in degrees Celsius. To convert the ADC result to a temperature value, you can use the following formula:

temperature = (ADC_result >> 4) * 0.0625;
This formula converts the 12-bit ADC result to a temperature value in degrees Celsius. The >> 4 operation shifts the bits of the ADC result to the right by 4 positions, effectively dividing the result by 16. The * 0.0625 operation multiplies the result by 0.0625, which is the scaling factor for the temperature sensor.

Regarding the ADC Configuration, I configured the ADC for reading analog input from a water level sensor. To ensure accurate readings from the sensor, it was essential to fine-tune the ADC's resolution, sampling period, and data alignment. Figure 2.7.1.3 is a snippet for the ADC configuration.

```
void ADC_Init(void) {
        // Enable ADC1 clock
        RCC->APB2ENR |= RCC_APB2ENR_ADC1EN;
        // Enable GPIOA clock
        RCC->AHB1ENR |= RCC_AHB1ENR_GPIOAEN;
        if (SENSOR == 1) {
                // Configure PA1 for ADC1 Channel 1
                GPIOA->MODER |= (PIN_SET << GPIO_MODER_MODE1_Pos);
                // Sample time configuration for Channel 1
                ADC1->SMPR2 |= (PIN_CONFIG << ADC_SMPR2_SMP1_Pos);
                // Regular sequence configuration for Channel 1
                ADC1->SQR3 |= (1 << ADC_SQR3_SQ1_Pos);
        }
        if (SENSOR == 2) {
                // Configure PA4 for ADC1 Channel 4
                GPIOA->MODER |= (PIN_SET << GPIO_MODER_MODE4_Pos);
                // Sample time configuration for Channel 4
                ADC1->SMPR2 |= (PIN_CONFIG << ADC_SMPR2_SMP4_Pos);
                // Regular sequence configuration for Channel 4
                ADC1->SQR3 |= (4 << ADC_SQR3_SQ1_Pos);
        }
        // Enable ADC1
        ADC1->CR2 |= ADC_CR2_ADON;
}
```

```
uint16_t Read_ADC(void) {
        ADC1->CR2 |= ADC_CR2_SWSTART;  // Start conversion for Channel 1
        while (!(ADC1->SR & ADC_SR_EOC))
                ;  // Wait for conversion to complete for Channel 1
        return (uint16_t) ADC1->DR;  // Read conversion result for Channel 1
}
```

Figure 2.7.1.3

The code is aimed at setting up and using an ADC in a microcontroller environment. It involves initializing the ADC and GPIO peripherals, configuring them based on the desired sensor input, and providing a means to read the analog-to-digital conversion results. This is a common task in embedded systems where digital processing of analog signals (i.e., sensor readings) is required.

2.7.2 General Purpose Input-Output

To do the GPIO Configuration, for determining the amount of moisture present in the soil, I initialized GPIO pins for LEDs which indicated the soil moisture level.

Each general-purpose I/O port has four 32-bit configuration registers (GPIOx_MODER, GPIOx_OTYPER, GPIOx_OSPEEDR and GPIOx_PUPDR), two 32-bit data registers (GPIOx_IDR and GPIOx_ODR), a 32-bit set/reset register (GPIOx_BSRR), a 32-bit locking register (GPIOx_LCKR) and two 32-bit alternate function selection register (GPIOx_AFRH and GPIOx_AFRL).

Table 1 given below is taken from “AN4899 STM32 Microcontrollers. It shows how the GPIO pins can be structured to be used as different peripherals.

Table 1. List of GPIO structures

Name		Abbreviation	Definition
Pin Type		S	Supply pin
		I	Input only pin
		I/O	Input / output pin
I/O structure		FT(1)	Five-volt tolerant I/O pin
		TT(1)	Three-volt tolerant I/O pin
		TC	Three-volt capable I/O pin (Standard 3.3 V I/O)
		B	Dedicated boot pin
		RST	Bidirectional reset pin with embedded weak pull-up resistor
Pin functions	Alternate functions	Functions selected through **GPIOx_AFR** registers	
	Additional functions	Functions directly selected and enabled through peripheral registers	

Most GPIO pins are not tolerant to 5v; they primarily support 3.3v. Regardless of how you intend to use a GPIO, it's essential to activate its clock. For GPIO output pins, you have the option to choose a speed that aligns with your needs, such as high-frequency switching. Thus, I have set up a GPIO pin for controlling the NPN 2222A transistor for relay control because the STM32 microcontroller pins were not able to provide enough output current which could be used for controlling the relay. I have connected the two water level sensors to pins PA1 and PA4 which read the analog signals from the sensors. Also, I have configured a timer for periodic ADC sampling and LED blinking patterns. Connected three LEDs to pins PB0, PB1 and PB2 and implemented logic to determine if the water level was low, normal, or high. Updated LED status based on soil moisture readings (different colors indicate different moisture levels). I have connected three resistors of 220 Ω each so that the LEDs don’t get burnt out by the current coming from the STM32 GPIO pins.

Also, the GPIO ports can be individually configured by software in several modes:
• Input floating
• Input pull-up
• Input-pull-down
• Analog
• Output open drain with pull-up or pull-down capability
• Output push-pull with pull-up or pull-down capability
• Alternate function push-pull with pull-up or pull-down capability

Figure 2.7.2.1 given below shows a code snippet from my code where the user can see the GPIO initialization for the LEDs, transistor, sensors, and the relay.

```
void GPIO_Init(void) {
        RCC->AHB1ENR |= RCC_AHB1ENR_GPIOBEN; // Enable clock for GPIOB

        // Configure PB0, PB1, and PB2 as output
        GPIOB->MODER |= (GPIO_MODER_MODER0_0 | GPIO_MODER_MODER1_0
                         | GPIO_MODER_MODER2_0);
        GPIOB->OTYPER &= ~(GPIO_OTYPER_OT0 | GPIO_OTYPER_OT1 | GPIO_OTYPER_OT2); // Push-pull
        GPIOB->OSPEEDR |= (GPIO_OSPEEDER_OSPEEDR0 | GPIO_OSPEEDER_OSPEEDR1
                         | GPIO_OSPEEDER_OSPEEDR2); // High speed
        GPIOB->PUPDR &=
                        ~(GPIO_PUPDR_PUPDR0 | GPIO_PUPDR_PUPDR1 | GPIO_PUPDR_PUPDR2); // No pull-up, pull-down

        RCC->AHB1ENR |= RCC_AHB1ENR_GPIODEN; // Enable clock for GPIOD (for PD12)

        // Configure PD12 as output
        GPIOD->MODER |= GPIO_MODER_MODER12_0;
        GPIOD->OTYPER &= ~GPIO_OTYPER_OT12; // Push-pull
        GPIOD->OSPEEDR |= GPIO_OSPEEDER_OSPEEDR12; // High speed
        GPIOD->PUPDR &= ~GPIO_PUPDR_PUPDR12; // No pull-up, pull-down
}
```

Figure 2.7.2.1

The relay control logic for managing water levels operates through a sequence of well-coordinated steps. Initially, low water level detection is essential; when the water dips below a set threshold, it triggers the system. This action sends a signal to a GPIO pin, which controls an NPN 2222A transistor. The transistor acts as a switch for the relay, either activating or deactivating it based on the received signal. Once activated, the relay powers the water pump, increasing the water level to the desired point. To prevent continuous operation and potential damage, the system includes a timer or feedback loop from the water level sensor, ensuring the pump runs only as needed. Additionally, the design incorporates a diode across the relay coil for back EMF protection. This diode is critical for safeguarding the circuit against voltage spikes when the relay is turned off, thereby maintaining the integrity and longevity of the system. This relay control logic thus ensures efficient water management while protecting the system from electrical mishaps.

2.7.3 MODBUS RTU Protocol

I have interconnected my second main hardware which is the Soil NPK Sensor with the Arduino UNO. The NPK sensor cannot be directly connected with the microcontroller, so I am using a MAX RS485 Transceiver module as the Soil NPK works on RS-485 Communication Protocol.
Figure 2.7.3.1 which is taken from "lastminuteengineers.com" explains the general connections I have done for the connecting a Soil NPK sensor with the Arduino UNO through a RS-485 connector.

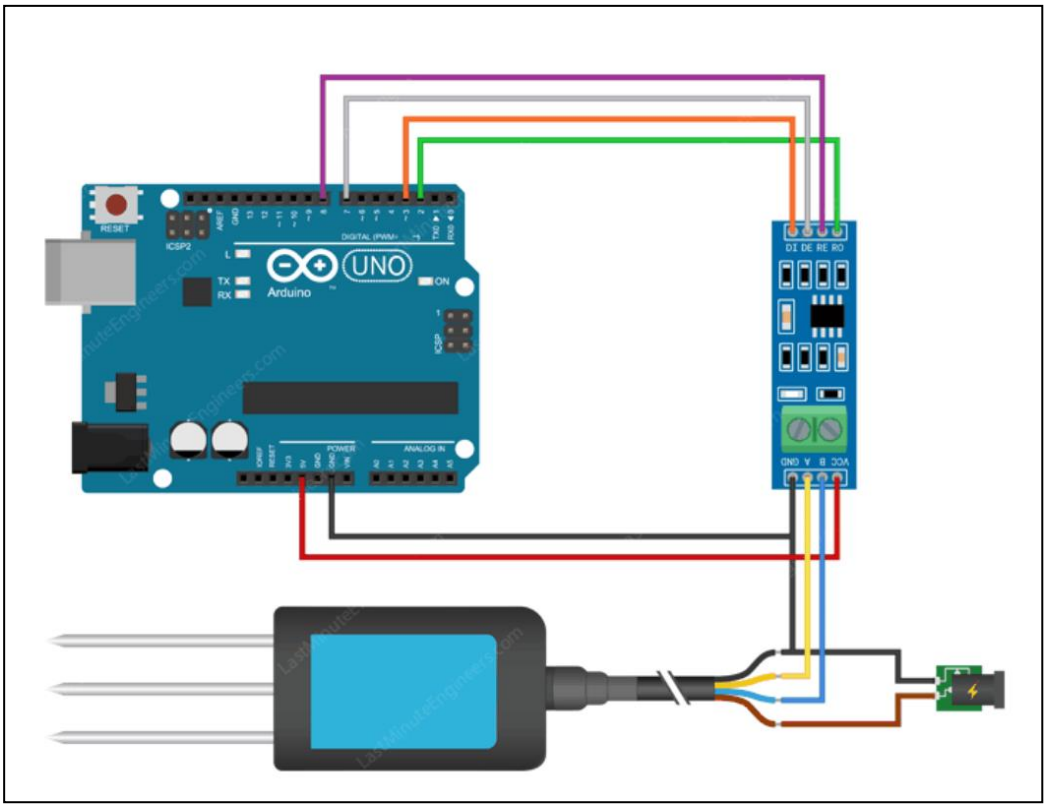


Figure 2.7.3.1

For the soil NPK sensor,

VCC is the VCC pin. Connects to 5V – 30V.
A is a differential signal that is connected to the A pin of the MAX485 Modbus Module.
B is another differential signal that is connected to the B pin of the MAX485 Modbus Module.
GND is the Ground pin.

RS485 is one of the most common methods of communicating with a device using the Modbus protocol. The RO pin connects to the RX pin of the UART, and the DI pin connects to the TX pin. The RE and DE Pins are responsible for setting the module in Receiver or Transmitter mode. When the RE pin is LOW and DE pin is LOW, the Module is set in the Receiver mode. When the DE pin is HIGH and RE pin is HIGH, the Module is set in the Transmitter mode. Pin A and Pin B are the output pins which carry the transmission signal.
To communicate with Arduino, we need an RS-485 transceiver module that converts a UART serial stream to RS-485. This module works on the Modbus RTU Protocol.

What exactly is Modbus Protocol?

Modbus is a widely used open serial communication protocol, initially released in 1979 by Modicon (now Schneider Electric), for data transmission between electronic devices over serial lines. It's a staple in automation, allowing networked devices like energy meters and humidity sensors to report data to supervisory computers or PLCs. Modbus comes in various forms, including RTU, ASCII, TCP, and Plus, and operates on a controller-peripheral architecture, using send-request and read-response messages. Originally on RS-232, it now predominantly uses RS-485 for its advantages in speed, distance, and network capacity. Modbus can communicate over RS-232/RS-485 or Ethernet, with RS-485 using two-wire connections for transmission. For networks with diverse devices, TCP/IP over Ethernet is better, allowing Modbus to coexist with other protocols. Modbus supports peer-to-peer, point-to-point, and multidrop networks and uses a controller-peripheral method for initiating transactions.

How does it Work?

Modbus communications follow a uniform structure, featuring four fundamental components in each message. This standardized format enables straightforward parsing of the content. Communication is always initiated by the controller, which sends a message to the peripheral. Upon receiving the message, the peripheral decodes and replies to it. Modbus utilizes specific functions to transmit instructions for reading and writing to the peripheral's internal memory registers. This is crucial for configuring and managing the peripheral's inputs and outputs. Each Modbus device usually has a register map indicating the locations for writing and reading configuration input and output data. For a comprehensive understanding of your device's functionality, consulting its register map is essential.

Modbus messages are structured as follows:

1) Device Address: Each peripheral device has a unique address it responds to when the controller addresses it. Messages not matching a device's address are disregarded. Device addresses range from 1 to 247. However, to avoid system instability, the number of devices should ideally not exceed 32 without additional hardware.

2) Function Code: In Modbus communication, the Function Code specifies whether the peripheral device should read or write data. It uses terminology derived from relay operation: a "coil" refers to a single-bit output, and a "discrete input" or "contact" refers to a single-bit input. Common Function Codes include:

    1. 01 READ COIL: Reads the status of coils.
    2. 02 READ DISCRETE INPUT: Reads the status of discrete inputs.
    3. 03 READ HOLDING REGISTERS: Reads holding registers.
    4. 04 READ INPUT REGISTERS: Reads input registers.
    5. 05 WRITE SINGLE COIL: Writes a single coil.
    6. 06 WRITE SINGLE REGISTER: Writes a single register.
    7. 15 WRITE MULTIPLE COILS: Writes multiple coils.
    8. 16 WRITE MULTIPLE REGISTERS: Writes multiple registers.

3) Data: The data field in a Modbus message contains either the data to be written or the data requested. In systems like the Arduino Modbus library, this field requires the starting register and the number of registers to be read or written.

4) CRC Error Check: CRC, or Cyclic Redundancy Check, is a method used for detecting errors in digital data, especially in digital communication networks. It adds redundancy (extra data) to the message for error checking without conveying additional information. The Modbus protocol uses a CRC check where two bytes are added to each message. Both the sending and receiving devices calculate the CRC. A mismatch in CRC values between the sender and receiver indicates an error in the transmitted message.

Figure 2.7.3.2 taken from gives a better understanding of the registers in the RTU protocol.

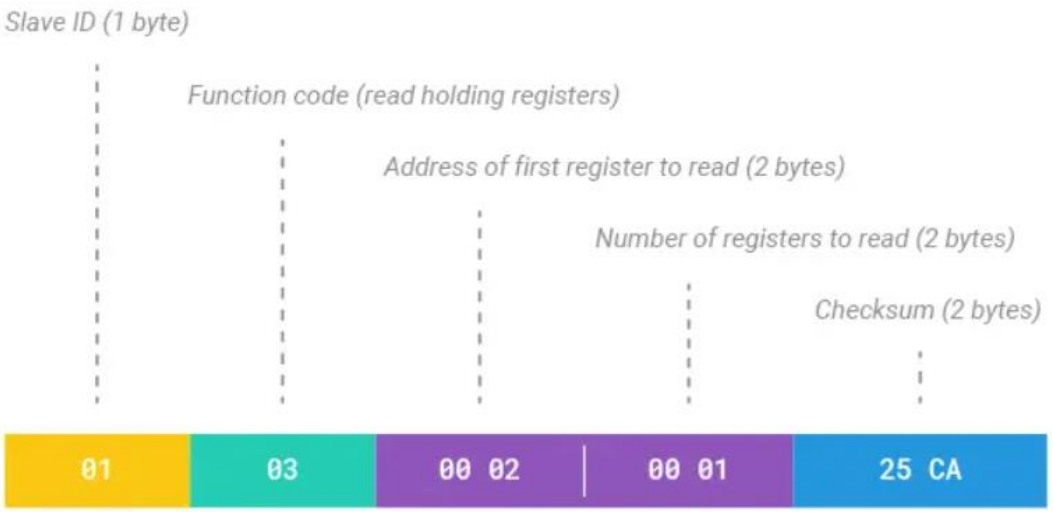


Figure 2.7.3.2

To read the values of nitrogen, phosphorous and potassium from the soil NPK Sensor, we must use the Modbus Protocol to read the registers from sensor using the RS-485 to TTL converter.

4.4.1 Read the Soil NPK Value from Device Address 0x01

Inquiry Frame

Address Code	Function Code	Register Start Address	Register Length	CRC_L	CRC_H
0x01	0x03	0x00 0x1E	0x00 0x03	0x34	0x0D

Answer Frame

Address Code	Function Code	Effective number of bytes	Nitrogen content	Phosphorus content	Potassium content	CRC_L	CRC_H
0x01	0x03	0x06	0x00 0x20	0x00 0x25	0x00 0x30	0x5A	0x3D

Figure 2.7.3.3

From Figure 2.7.3.3 which is given above taken from the “JXCT Soil NPK Sensor Datasheet”, we must read values for the nitrogen, phosphorous and potassium content starting from register address 0x001E and read three registers. The values are calibrated by converting to decimal equivalent as follows:

NPK content:
0020 H（hexadecimal）=32=>Nitrogen=32mg/kg
0025 H（hexadecimal）=37=>Phosphorus=37mg/kg
0030 H（hexadecimal）=48=>Potassium=48mg/kg

Other method of getting the values is separately reading out the contents of Nitrogen, Phosphorous and Potassium as shown in Figure 2.7.3.4:

4.4.2 Reading the value of soil nitrogen at device address 0x01

Inquiry Frame

Address Code	Function Code	Register Start Address	Register Length	CRC_L	CRC_H
0x01	0x03	0x00 0x1e	0x00 0x01	0xB5	0xCC

Answer Frame

Address Code	Function Code	Effective No. of bytes	Nitrogen content	CRC_L	CRC_H
0x01	0x03	0x02	0x00 0x20	0x5A	0x3D

Nitrogen content : 0020 H (hexadecimal) =32=>Nitrogen=32mg/kg

4.4.3 Reading the Value of Soil Phosphorus in Device Address 0x01

Inquiry Frame

Address Code	Function Code	Register start address	Register length	CRC_L	CRC_H
0x01	0x03	0x00 0x1f	0x00 0x01	0xE4	0x0C

Answer Frame

Address Code	Function Code	Effective No. of bytes	Phosphorus content	CRC_L	CRC_H
0x01	0x03	0x02	0x00 0x25	0x5A	0x3D

Phosphorus content : 0025 H (16hexadecimal) =37=>Phosphorus=37mg/kg

4.4.4 Reading the Value of Soil Potassium in Device Address 0x01

Inquiry Frame

Address Code	Function Code	Register start address	Register length	CRC_L	CRC_H
0x01	0x03	0x00 0x20	0x00 0x01	0x85	0xC0

Answer Frames

Address Code	Function Code	Effective No. of bytes	Potassium content	CRC_L	CRC_H
0x01	0x03	0x02	0x00 0x30	0x5A	0x3D

Potassium : 0030 H (16hexadecimal) =48=>Potassium=48mg/kg

Figure 2.7.3.4

Figure 2.7.3.5 shows code snippet from my code and the terminal window where the contents of Nitrogen, Phosphorous and Potassium can be seen which are being read from the sensor.

```
void loop() {
    uint8_t val1, val2, val3;

    val1 = readSensor(nitro, sizeof(nitro));
    _delay_ms(250);

    val2 = readSensor(phos, sizeof(phos));
    _delay_ms(250);

    val3 = readSensor(pota, sizeof(pota));
    _delay_ms(250);

    // Print Nitrogen value
    Serial.print("Nitrogen: ");
    Serial.print(val1);
    Serial.println(" mg/kg");

    // Print Phosphorus value
    Serial.print("Phosphorus: ");
    Serial.print(val2);
    Serial.println(" mg/kg");

    // Print Potassium value
    Serial.print("Potassium: ");
    Serial.print(val3);
    Serial.println(" mg/kg");

    _delay_ms(1000);
}
```

```
Nitrogen: 104 mg/kg
Phosphorous: 37 mg/kg
Potassium: 52 mg/kg


Nitrogen: 104 mg/kg
Phosphorous: 37 mg/kg
Potassium: 52 mg/kg
```

Figure 2.7.3.5

2.8 Testing Process

The testing phase of the project was comprehensive, primarily focusing on ensuring the integrity of electronic components in a water-intensive environment. Key challenges included protecting the electronics from water damage and addressing the susceptibility of water sensors to corrosion. Regular recalibration was necessary to maintain the accuracy of these sensors, especially when interfacing with soil. Additionally, the relay and water pump required an external power source, for which I utilized the power supply from the ESD lab's parts kit, connecting it via a power jack. In contrast, the soil NPK sensor proved to be robust, consistently delivering accurate readings without any significant maintenance issues.

3 RESULTS AND ERROR ANALYSIS

The project's anticipated results, as outlined in my proposal, were substantially achieved with some notable learning experiences. Below are a few points about how I tackled with the errors and other analysis.

- Successful Sensor Integration: Achieved interfacing of the water level sensor with the STM32F411e microcontroller.

- Relay and Water Pump Control: Used a relay as a switch to control the water pump, effectively managing the system.

- ADC Learning Curve: Gained knowledge in using the Analog-to-Digital Converter (ADC) for reading sensor values.

- Adapting to Technical Challenges:
  - Initially faced issues with inaccurate sensor readings using an interrupt-based method.
  - Switched to a polling approach for ADC initialization for more reliable readings.

- Relay Driving Issue and Solution:
  - Encountered a problem with insufficient current from STM32 GPIO pins to drive the relay.
  - Resolved by using a transistor to amplify the current, enabling efficient relay control.

- Partial Success with Soil NPK Sensor:
  - Struggled to implement Modbus protocol on STM32F411e.
  - Opted to interface the sensor with Arduino UNO due to better support and reference codes.

- Modbus Protocol Exploration:
  - Managed partial implementation on STM32: could send requests but not receive responses.
  - Gained extensive knowledge in RS-485 communication and Modbus RTU Protocol.

- Insight on Early Research Importance: Recognized that earlier research on using Modbus protocol with STM32 Microcontrollers might have improved results.

- Overall Project Outcome:
  - While the outcome was not as perfect as initially intended, the project was a valuable learning experience.
  - Demonstrated practical functionality and adaptability in addressing and overcoming technical challenges.

Figure 3.1 given below from the “Intronix LogicPort Logic Analyzer Application” shows the signals where I was able to send data from the STM32F411e to the soil NPK sensor through the Modbus Protocol.

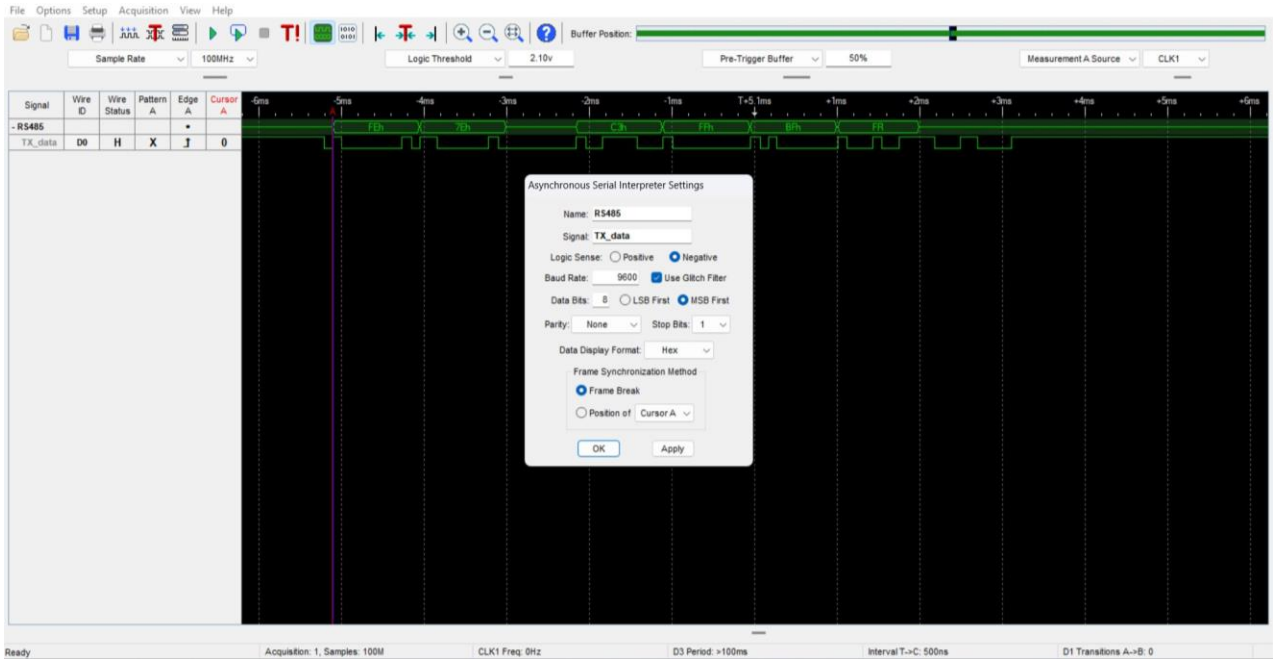


Figure 3.1

4 CONCLUSION

I selected this topic because it offered me the opportunity to engage extensively with various firmware, enhancing my comprehension of both software and hardware elements within the project. Furthermore, during my undergraduate studies, I collaborated on a similar project with a team of six, which inspired me to challenge myself and see if I could successfully complete a similar project independently. While the software aspect of the project was straightforward, I particularly enjoyed working with the hardware components. Additionally, this system offers several practical benefits in real-world applications which is as follows:

The development of an agricultural automated irrigation system leads to increased water efficiency, improved crop health, cost savings, adaptability to different environmental conditions, data-driven agricultural practices, and environmental benefits. It represents a significant advancement in sustainable and efficient farming, offering a versatile and customizable solution for various agricultural needs.

5 FUTURE DEVELOPMENT IDEAS

The future scope for enhancing this agricultural technology includes several ambitious and practical upgrades:

- LCD or OLED Display Integration: The system will be equipped with either an LCD or OLED display. This display will serve as a user interface, providing real-time information on the soil's nutrient content, specifically the levels of Nitrogen (N), Phosphorus (P), and Potassium (K). This feature will allow farmers to monitor the soil health quickly and efficiently, making informed decisions about fertilization and crop management.

- Inclusion of pH and Temperature Sensors: To gain a more comprehensive understanding of the soil's condition, the system will incorporate advanced sensors capable of detecting the soil's pH and temperature. The pH sensor will measure the acidity or alkalinity of the soil, which is crucial for plant growth, while the temperature sensor will monitor the soil temperature, an important factor affecting seed germination and root development. This data will further assist in optimizing the growing conditions for various crops.

- Implementation of a GSM Module: By integrating a GSM (Global System for Mobile Communications) module, the system will facilitate remote data access. This means that farmers can receive updates on soil conditions directly on their smartphones, making it convenient to monitor their fields from any location. The GSM module will transmit data such as NPK values, pH levels, and temperature readings, providing a comprehensive overview of the soil's health in real-time.

- Development of an Automatic Water Pump Control Interface: A significant upgrade will be the implementation of an automatic control system for the water pump. This system will be based on the data received from the soil sensors. For instance, if the moisture level in the soil is low, the system can automatically activate the water pump to irrigate the field. This feature aims to optimize water usage and reduce manual labor, ensuring that the crops receive the right amount of water at the right time.

These enhancements will not only improve the efficiency of agricultural practices but also support sustainable farming methods by providing precise data for better resource management.

6 ACKNOWLEDGEMENTS

I extend my heartfelt thanks to Professor Dr. Linden McClure for his invaluable guidance and teaching throughout the course.

My gratitude also goes to the teaching assistants whose support was crucial during the project implementation phase.

Additionally, I am grateful to my peers for their collaborative spirit and the insightful ideas they shared, which greatly assisted me in managing the project on my own.

7 REFERENCES

[1] Wikipedia , “Irrigation Controller” https://en.wikipedia.org/wiki/Irrigation_controller

[2] Wikipedia, “Irrigation sprinkler” https://en.wikipedia.org/wiki/Irrigation_sprinkler

[3] Energypedia, “Smart Irrigation Controls” https://energypedia.info/wiki/Smart_Irrigation_Controls_(PA_Technology)

[4] Sciencedirect “Smart Irrigation Systems using IoT” https://www.sciencedirect.com/science/article/pii/S2772427122000791

[5] Controllers Tech “RS485 and Modbus with STM32” https://controllerstech.com/rs485-module-and-stm32/

[6] Last Minute Engineers “Water Sensor with Arduino” https://lastminuteengineers.com/water-level-sensor-arduino-tutorial/

[7] Last Minute Engineers “Soil NPK Sensor with Arduino” https://lastminuteengineers.com/soil-npk-sensor-arduino-tutorial/

[8] Arduino Modbus https://docs.arduino.cc/learn/communication/modbus

[9] Instructables “Smart Watering Plant in STM32” https://www.instructables.com/Smart-Watering-Plant-Using-RT-Thread-RTOS-in-STM32/

[10] RS485 Module with converter https://hobbycomponents.com/wired-wireless/663-max485-rs485-transceiver-module

[11] GitHub “Modbus Library for Arduino” https://github.com/arduino-libraries/ArduinoModbus/blob/master/README.adoc

[12] Modbus RTU protocol https://www.csimn.com/CSI_pages/Modbus101.html

[13] Last Minute Engineers “Relay with Arduino” https://lastminuteengineers.com/one-channel-relay-module-arduino-tutorial/

[14] Controllers Tech “ADC in STM32” https://controllerstech.com/stm32-adc-single-channel/

[15] YouTube “DIY Soil NPK meter” https://www.youtube.com/watch?v=vyb_qHuY6DA

[16] STM Microcontrollers “AN4899 GPIO pins” https://www.st.com/resource/en/application_note/an4899-stm32-microcontroller-gpio-hardware-settings-and-lowpower-consumption-stmicroelectronics.pdf

[17] DeepBlue “Stm32 ADC” https://deepbluembedded.com/stm32-adc-tutorial-complete-guide-with-examples/

[18] Automated Irrigation System project https://www.elprocus.com/automated-irrigation-system-project/

[19] Modbus PC interface application https://www.simplymodbus.ca/RTUSlaveManual8.htm

[20] ST Microelectronics “UART Configurations” https://wiki.st.com/stm32mcu/wiki/Getting_started_with_UART

[21] https://www.youtube.com/watch?v=GyXe9BezQmg

[22] https://www.youtube.com/watch?v=Gn1UhtXqiaI

[23] https://www.youtube.com/watch?v=90DtjtchKj0

[24] https://www.youtube.com/watch?v=txi2p5_OjKU

[25] GitHub “Greenhouse CO2 Controller” https://github.com/ussaka/Greenhouse-co2-controller/blob/main/co2_controller/src/modbus/ModbusMaster.cpp

8 APPENDICES

Several appendices have been attached to this report in the order shown below.

8.1 Appendix - Bill of Materials

Part Description	Source	Cost
Water Level Sensor	Digi-Key www.digikey.com	$3.95
Soil NPK Sensor	Amazon www.amazon.com	$40.00
STM32F411E	Embedded Systems Parts Kit	$0.00
Arduino UNO	Amazon www.amazon.com	$27.60
JZC-11F Relay	ITLL Laboratory	$0.00
Water Pump 12V DC	Amazon www.amazon.com	$12.00
PN2222A	Embedded Systems Parts Kit	$0.00
Connecting Wires	Amazon www.amazon.com	$6.00
TOTAL		$89.55

8.2 Appendix – Schematics

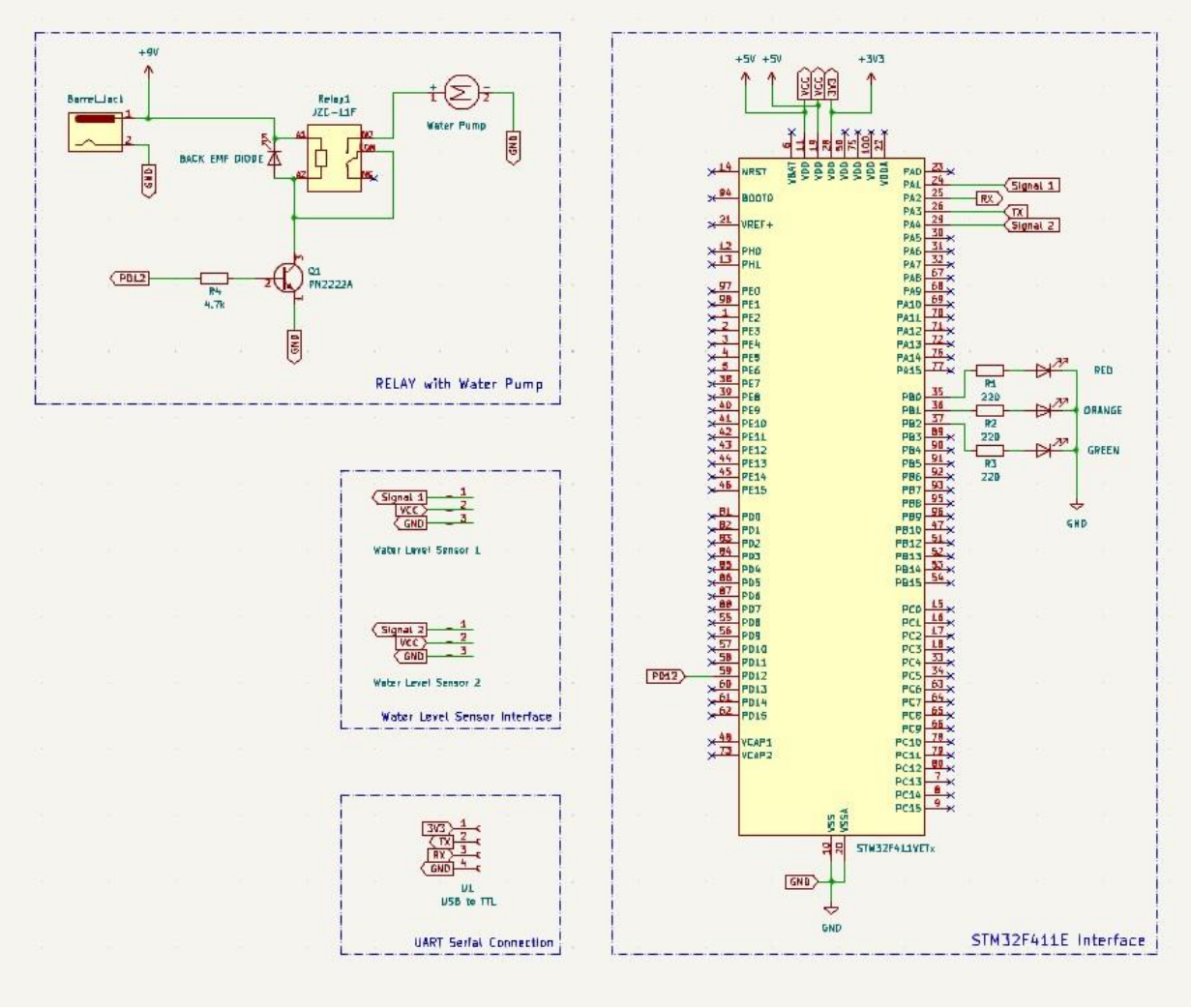


Figure 8.2.1

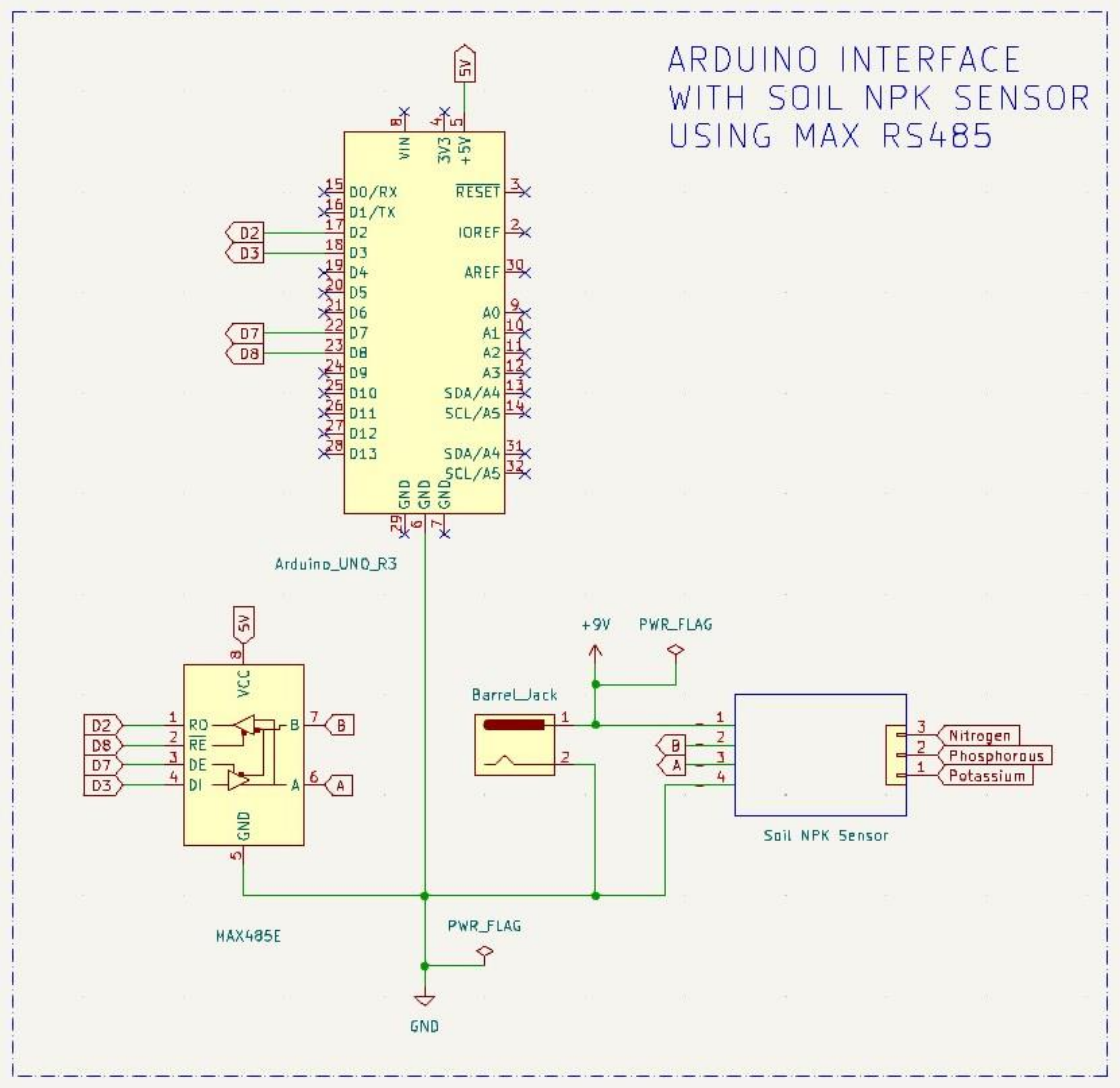

ARDUINO INTERFACE
WITH SOIL NPK SENSOR
USING MAX RS485
5V
VIN
3V3
+5V
D0/RX
D1/TX
D2
D3
D4
D5
D6
D7
D8
D9
D10
D11
D12
D13
RESET
IOREF
AREF
A0
A1
A2
A3
SDA/A4
SCL/A5
GND
Arduino_UNO_R3
VCC
RO
RE
DE
DI
B
A
MAX485E
+9V
PWR_FLAG
Barrel_Jack
Soil NPK Sensor
Nitrogen
Phosphorous
Potassium

Figure 8.2.2

8.3 Appendix - Firmware Source Code

8.3.1 Water Level Sensor with relay and water pump code

Main.c

```
/*
 * main.c
 *
 *     Application Point of Entry
 *
 *  Created on: Dec 17, 2023
 *      Author: Unmesh
 */

#include <ADC_Config.h>
#include <definitions.h>
#include <LED_Relay_Control.h>
#include <UART_Terminal.h>

int main(void) {
	// Initialize UART, ADC and GPIOs
	UART2_Init();
	ADC_Init();
	GPIO_Init();
	char myString[] = "Start moisture detection:\r\n";
	UART2_SendString(myString);                    // Start displaying on the
terminal window

	char buffer[50];                               // Buffer for storing
value

	while (1) {
		// Read the value from the sensor
		if (SENSOR == 1) {
			uint16_t sensorValue = Read_ADC();
			// Convert sensor value to a string
			sprintf(buffer, "Sensor 1 value is %d\r\n", sensorValue);
			// Send the string over UART
			UART2_SendString(buffer);
			// Control the LEDs and Relay
			LED_Control(sensorValue);
		}

		if (SENSOR == 2) {
			uint16_t sensorValue = Read_ADC();
			// Convert sensor value to a string
			sprintf(buffer, "Sensor 2 value is %d\r\n", sensorValue);
			// Send the string over UART
			UART2_SendString(buffer);
			// Control the LEDs and Relay
			LED_Control(sensorValue);
		}

		// Delay
		for (uint32_t i = 0; i < 500000; i++)
			;
	}
}
```

ADC_Config.h

```c
/*
 * ADC_Config.h
 *
 *     This header file declares the functions for initializing and
reading from the ADC (Analog-to-Digital Converter).
 *     It provides the declarations for ADC_Init and Read_ADC functions.
 *     ADC_Init is responsible for initializing the ADC peripheral and
configuring the associated GPIO pins.
 *     Read_ADC handles the process of starting an ADC conversion,
waiting for its completion, and then reading the converted value.
 *     This file is a part of the ADC configuration module, specifically
tailored for use with STM32 microcontrollers.
 *     It should be included in any module that requires interaction
with the ADC peripheral.
 *
 *  Created on: Dec 16, 2023
 *      Author: Unmesh
 */

#ifndef INC_ADC_CONFIG_H_
#define INC_ADC_CONFIG_H_

#include "stdint.h"

void ADC_Init(void);

uint16_t Read_ADC(void);


#endif /* INC_ADC_CONFIG_H_ */
```

ADC_Config.c

```c
/*
 * ADC_Config.c
 *
 *     This file contains the implementation for the initialization and reading
of ADC values.
 *     It includes functions to set up the ADC peripheral and read data from it.
 *     The ADC_Init function initializes the ADC and configures the necessary
GPIO pins.
 *     It supports configuration for different sensors by checking the SENSOR
macro.
 *     The Read_ADC function starts the ADC conversion, waits for it to
complete, and then returns the result.
 *     This code is designed for STM32 microcontrollers and utilizes specific
registers and macros for configuration.
 *
 *  Created on: Dec 16, 2023
 *      Author: Unmesh
 */
```

```c
#include <ADC_Config.h>
#include <definitions.h>

void ADC_Init(void) {
	// Enable ADC1 clock
	RCC->APB2ENR |= RCC_APB2ENR_ADC1EN;
	// Enable GPIOA clock
	RCC->AHB1ENR |= RCC_AHB1ENR_GPIOAEN;

	if (SENSOR == 1) {
		// Configure PA1 for ADC1 Channel 1
		GPIOA->MODER |= (PIN_SET << GPIO_MODER_MODE1_Pos);
		// Sample time configuration for Channel 1
		ADC1->SMPR2 |= (PIN_CONFIG << ADC_SMPR2_SMP1_Pos);
		// Regular sequence configuration for Channel 1
		ADC1->SQR3 |= (1 << ADC_SQR3_SQ1_Pos);
	}

	if (SENSOR == 2) {
		// Configure PA4 for ADC1 Channel 4
		GPIOA->MODER |= (PIN_SET << GPIO_MODER_MODE4_Pos);
		// Sample time configuration for Channel 4
		ADC1->SMPR2 |= (PIN_CONFIG << ADC_SMPR2_SMP4_Pos);
		// Regular sequence configuration for Channel 4
		ADC1->SQR3 |= (4 << ADC_SQR3_SQ1_Pos);
	}

	// Enable ADC1
	ADC1->CR2 |= ADC_CR2_ADON;
}

uint16_t Read_ADC(void) {
	ADC1->CR2 |= ADC_CR2_SWSTART;  // Start conversion for Channel 1
	while (!(ADC1->SR & ADC_SR_EOC))
		;  // Wait for conversion to complete for Channel 1
	return (uint16_t) ADC1->DR;  // Read conversion result for Channel 1
}
```

Led_Relay_Control.h

```c
/*
 * LED_Relay_Control.h
 *
 *     This header file provides declarations for functions used in controlling
LEDs and a relay based on sensor readings.
 *     It includes the declaration of GPIO_Init for initializing GPIO pins and
LED_Control for managing the state of LEDs and a relay.
 *     The functions cater to applications requiring sensor-driven visual
indicators and relay operation.
 *
 *  Created on: Dec 16, 2023
 *      Author: Unmesh
 */
#ifndef INC_LED_RELAY_CONTROL_H_
#define INC_LED_RELAY_CONTROL_H_

#include "stdint.h"

void GPIO_Init(void);

void LED_Control(uint16_t sensorValue);
#endif /* INC_LED_RELAY_CONTROL_H_ */
```

Led_Relay_Control.c

```c
/*
 * LED_Relay_Control.c
 *
 *     This file contains the implementation for controlling LEDs and a relay
based on sensor values.
 *     It includes GPIO_Init to set up GPIO pins for LED and relay control, and
LED_Control to manage LEDs and relay state.
 *     The LED_Control function uses sensor values to determine which LED to
turn on and whether to activate the relay.
 *     Suitable for applications requiring sensor-based visual indicators and
relay control.
 *
 *  Created on: Dec 16, 2023
 *      Author: Unmesh
 */
#include <definitions.h>

void GPIO_Init(void) {
	RCC->AHB1ENR |= RCC_AHB1ENR_GPIOBEN; // Enable clock for GPIOB

	// Configure PB0, PB1, and PB2 as output
	GPIOB->MODER |= (GPIO_MODER_MODER0_0 | GPIO_MODER_MODER1_0
			| GPIO_MODER_MODER2_0);
	GPIOB->OTYPER &= ~(GPIO_OTYPER_OT0 | GPIO_OTYPER_OT1 | GPIO_OTYPER_OT2);
// Push-pull
	GPIOB->OSPEEDR |= (GPIO_OSPEEDER_OSPEEDR0 | GPIO_OSPEEDER_OSPEEDR1
			| GPIO_OSPEEDER_OSPEEDR2); // High speed
	GPIOB->PUPDR &=
			~(GPIO_PUPDR_PUPDR0 | GPIO_PUPDR_PUPDR1 |
GPIO_PUPDR_PUPDR2); // No pull-up, pull-down

	RCC->AHB1ENR |= RCC_AHB1ENR_GPIODEN; // Enable clock for GPIOD (for PD12)

	// Configure PD12 as output
	GPIOD->MODER |= GPIO_MODER_MODER12_0;
	GPIOD->OTYPER &= ~GPIO_OTYPER_OT12; // Push-pull
	GPIOD->OSPEEDR |= GPIO_OSPEEDER_OSPEEDR12; // High speed
	GPIOD->PUPDR &= ~GPIO_PUPDR_PUPDR12; // No pull-up, pull-down
}

void LED_Control(uint16_t sensorValue) {
	// Turn off all LEDs
	GPIOB->BSRR = GPIO_BSRR_BR0 | GPIO_BSRR_BR1 | GPIO_BSRR_BR2;

	if (sensorValue >= 0 && sensorValue < 1700) {
		// Turn on LED connected to PB0
		GPIOB->BSRR = GPIO_BSRR_BS0;
	} else if (sensorValue >= 1700 && sensorValue < 3000) {
		// Turn on LED connected to PB1
		GPIOB->BSRR = GPIO_BSRR_BS1;
	} else {
		// Turn on LED connected to PB2
		GPIOB->BSRR = GPIO_BSRR_BS2;
	}

	GPIOD->BSRR = GPIO_BSRR_BR12;

	if (sensorValue > 500 && sensorValue < 3200) {
		// Turn on the Relay connected to PD12
		GPIOD->BSRR = GPIO_BSRR_BS12;
	}
}
```

UART_Terminal.h

```
/*
 * UART_Terminal.h
 *
 *     This header file defines the interface for UART communication
functionalities.
 *     It declares UART2_Init for initializing the UART peripheral and
UART2_SendString for transmitting strings via UART.
 *     These functions are essential for serial communication tasks in embedded
systems, including data transmission and receiving sensor information.
 *
 *  Created on: Dec 16, 2023
 *      Author: Unmesh
 */

#ifndef INC_UART_TERMINAL_H_
#define INC_UART_TERMINAL_H_

void UART2_Init(void);

void UART2_SendString();

#endif /* INC_UART_TERMINAL_H_ */
```

UART_Terminal.c

```
/*
 * UART_Terminal.c
 *
 *     This file implements functions for initializing and handling UART
communications.
 *     It includes UART2_Init for setting up USART2 with the appropriate baud
rate and interrupt configuration.
 *     UART2_SendString is used to transmit strings over UART.
 *     This code is designed for systems requiring serial communications, such
as interfacing with sensors or terminal I/O.
 *
 *  Created on: Dec 16, 2023
 *      Author: Unmesh
 */

#include <definitions.h>
#include <UART_Terminal.h>

void UART2_Init(void) {
	RCC->AHB1ENR |= RCC_AHB1ENR_GPIOAEN; // Enable clocks for GPIOA and
USART2
	RCC->APB1ENR |= RCC_APB1ENR_USART2EN;

	GPIOA->MODER |= GPIO_MODER_MODER2_1 | GPIO_MODER_MODER3_1; // Set pins
PA2 and PA3 in alternate function mode for USART2
	GPIOA->AFR[0] |= (0x7 << 8) | (0x7 << 12);

	USART2->BRR = 0x683;                                                //
Baud Rate Configuration
	USART2->CR1 |= USART_CR1_TE | USART_CR1_RE | USART_CR1_UE; // Enable
transmitter, receiver, and USART2 module

	USART2->CR1 |= USART_CR1_RXNEIE; // Enable interrupt on receive data
register
	NVIC_EnableIRQ(USART2_IRQn);                                    // Enable
USART2 interrupt
}
```

```
void UART2_SendString(char *str)        // Send string via UART2
{
       while (*str) {
              while (!(USART2->SR & USART_SR_TXE))
                     ;          // Wait until transmit data register is empty
              USART2->DR = *str++;
       }
}
```

Definitions.h

```
/*
 * definitions.h
 *
 *  In this file, the pin configurations and other definitions are included
 *
 *  Created on: Dec 16, 2023
 *      Author: Unmesh
 */

#ifndef INC_DEFINITIONS_H_
#define INC_DEFINITIONS_H_

#include "stm32f4xx.h"
#include <stdio.h>

#define SENSOR                      2
#define PIN_SET          0b11
#define PIN_CONFIG       0b011


#endif /* INC_DEFINITIONS_H_ */
```

8.3.2 Soil NPK Sensor with Arduino Code

```
/*
 * Soil NPK sensor with Arduino
 *
 * This program is designed for Arduino to interface with a soil NPK
(Nitrogen, Phosphorus, Potassium) sensor. It utilizes
 * Modbus RTU protocol for sensor communication, and RS485 for data
transmission. The program configures the USART for
 * serial communication, sets up necessary pins for RS485 module control,
and processes the sensor data to read the NPK
 * values. Specific commands are sent to the sensor to retrieve nitrogen,
phosphorus, and potassium levels, which are then
 * displayed through the Arduino's serial monitor. Essential delays are
included for stabilization and smooth display of
 * data. This code is a useful resource for anyone looking to monitor soil
nutrient levels using Arduino technology.
 *
 *  Created on: Dec 17, 2023
 *      Author: Unmesh
 */
```

```
#include <avr/io.h>
#include <util/delay.h>
#include <SoftwareSerial.h>

// Define the clock frequency and baud rate for serial communication
#define F_CPU 16000000UL // Clock frequency: 16MHz
#define BAUD 9600 // Baud rate for serial communication
#define MYUBRR F_CPU/16/BAUD-1 // Formula to calculate the UBRR value

// Definitions for RS485 module control pins (RE and DE)
#define RE_DDR DDRB // Data Direction Register for RE pin
#define RE_PORT PORTB // Port for RE pin
#define RE_PIN PB0 // Pin number for RE
#define DE_DDR DDRB // Data Direction Register for DE pin
#define DE_PORT PORTB // Port for DE pin
#define DE_PIN PB1 // Pin number for DE

// Modbus RTU requests for reading NPK values
const uint8_t nitro[] = {0x01, 0x03, 0x00, 0x1e, 0x00, 0x01, 0xe4, 0x0c};
const uint8_t phos[] = {0x01, 0x03, 0x00, 0x1f, 0x00, 0x01, 0xb5, 0xcc};
const uint8_t pota[] = {0x01, 0x03, 0x00, 0x20, 0x00, 0x01, 0x85, 0xc0};

uint8_t values[11];

// Function to initialize USART for serial communication
void USART_Init(unsigned int ubrr) {
    // Set baud rate
    UBRR0H = (uint8_t)(ubrr>>8);
    UBRR0L = (uint8_t)ubrr;
    // Enable receiver and transmitter
    UCSR0B = (1<<RXEN0)|(1<<TXEN0);
    // Set frame format: 8 data, 2 stop bit
    UCSR0C = (1<<USBS0)|(3<<UCSZ00);
}

// Function to transmit data via USART
void USART_Transmit(uint8_t data) {
    // Wait for empty transmit buffer
    while (!(UCSR0A & (1<<UDRE0)));
    // Put data into buffer, sends the data
    UDR0 = data;
}


// Function to receive data via USART
```

```
uint8_t USART_Receive(void) {
    // Wait for data to be received
    while (!(UCSR0A & (1<<RXC0)));
    // Get and return received data from buffer
    return UDR0;
}

// Setup function for initial configurations
void setup() {
    // Initialize the Serial communication
    Serial.begin(BAUD);

    // Set RE and DE as output pins
    RE_DDR |= (1<<RE_PIN);
    DE_DDR |= (1<<DE_PIN);

    // Initial delay for stabilization
    _delay_ms(500);
}

// Function to read sensor data
uint8_t readSensor(const uint8_t *command, uint8_t commandLength) {
    // Set RS485 to transmit mode
    DE_PORT |= (1<<DE_PIN);
    RE_PORT |= (1<<RE_PIN);
    _delay_ms(10);

    // Send the command
    for (uint8_t i = 0; i < commandLength; i++) {
        USART_Transmit(command[i]);
    }

    // Set RS485 to receive mode
    DE_PORT &= ~(1<<DE_PIN);
    RE_PORT &= ~(1<<RE_PIN);

    // Read the response
    for (uint8_t i = 0; i < 7; i++) {
        values[i] = USART_Receive();
    }

    // Return the sensor value
    return values[4];
}
```

```
// Main loop function
void loop() {
    uint8_t val1, val2, val3;

    // Read Nitrogen value
    val1 = readSensor(nitro, sizeof(nitro));
    _delay_ms(250);

    // Read Phosphorous value
    val2 = readSensor(phos, sizeof(phos));
    _delay_ms(250);

    // Read Potassium value
    val3 = readSensor(pota, sizeof(pota));
    _delay_ms(250);

    // Print Nitrogen value
    Serial.print("Nitrogen: ");
    Serial.print(val1);
    Serial.println(" mg/kg");

    // Print Phosphorus value
    Serial.print("Phosphorus: ");
    Serial.print(val2);
    Serial.println(" mg/kg");

    // Print Potassium value
    Serial.print("Potassium: ");
    Serial.print(val3);
    Serial.println(" mg/kg");

    // Delay for displaying smoothly
    _delay_ms(1000);
}

// Main initialization functions
int main() {
    init();
    setup();
    while (1) {
        loop();
    }
}
```

8.3.3 Soil NPK Sensor with STM32 Code

Main.c

```c
/*
 * main.c
 *
 *     Application Point of Entry
 *
 *  Created on: Dec 17, 2023
 *      Author: Unmesh
 */

#include "stm32f4xx.h"
#include <stdio.h>
#include "Clock_config.h"
#include "UART_Data.h"

int main(void) {
    Clock_Init();
    UART2_Init();

    processElement((uint8_t*)nitro, "Nitrogen");
    processElement((uint8_t*)phos, "Phosphorus");
    processElement((uint8_t*)pota, "Potassium");

    while (1) {
    }
}
```

Clock_config.h

```c
/*
 * Clock_config.h
 *
 *     This header file declares functions for system clock configuration and
implementing a delay mechanism.
 *     It provides the interface for Clock_Init, which sets up the
microcontroller's main clock, and delay_ms, a function for creating software
delays.
 *     These functions are crucial for initializing and managing timing
functionalities in embedded systems, especially for STM32F4 series
microcontrollers.
 *
 *  Created on: Dec 17, 2023
 *      Author: Unmesh
 */

#ifndef SRC_CLOCK_CONFIG_H_
#define SRC_CLOCK_CONFIG_H_

#include "stdint.h"

void Clock_Init(void);

void delay_ms(uint32_t ms);


#endif /* SRC_CLOCK_CONFIG_H_ */
```

Clock_config.c

```c
/*
 * Clock_config.c
 *
 *     This file contains the implementation for the system clock configuration
and a simple delay function.
 *     Clock_Init configures the system clock using the High-Speed External
(HSE) oscillator and sets up the PLL for desired frequency.
 *     The delay_ms function provides a simple software delay mechanism.
 *     Essential for setting up the system clock for STM32F4 series
microcontrollers to ensure optimal performance and timing accuracy.
 *
 *  Created on: Dec 17, 2023
 *      Author: Unmesh
 */

#include "stm32f4xx.h"
#include <stdio.h>
#include "Clock_config.h"

void Clock_Init(void) {
	RCC->CR |= RCC_CR_HSEON;
	while (!(RCC->CR & RCC_CR_HSERDY))
		;
	RCC->APB1ENR |= RCC_APB1ENR_PWREN;
	PWR->CR |= PWR_CR_VOS;
	FLASH->ACR |= FLASH_ACR_ICEN | FLASH_ACR_DCEN | FLASH_ACR_LATENCY_3WS;
	RCC->PLLCFGR = (RCC_PLLCFGR_PLLSRC_HSE | 12 | (192 << 6) | (1 << 16) | (4
<< 24));

	RCC->CR |= RCC_CR_PLLON;
	while (!(RCC->CR & RCC_CR_PLLRDY))
		;
	RCC->CFGR &= ~RCC_CFGR_SW;
	RCC->CFGR |= RCC_CFGR_SW_PLL;
	while ((RCC->CFGR & RCC_CFGR_SWS) != RCC_CFGR_SWS_PLL)
		;
}

void delay_ms(uint32_t ms) {
    volatile uint32_t count = 24000 * ms;
    while (count--) {
    }
}
```

UART_Data.h

```c
/*
 * UART_Data.h
 *     This header file defines the interface for UART communication
functionalities in STM32F4xx microcontrollers.
 *     It includes declarations for UART2 initialization, sending strings,
handling data transmission and reception, and processing elements.
 *     Additionally, it declares Modbus RTU requests for reading NPK (Nitrogen,
Phosphorus, Potassium) values, crucial for applications in precision
agriculture or similar fields.
 *     These functions are fundamental for projects that require robust and
efficient serial communication, such as sensor data transmission or remote
command processing.
 *
 *  Created on: Dec 17, 2023
 *      Author: Unmesh
 */
```

```c
#ifndef SRC_UART_DATA_H_
#define SRC_UART_DATA_H_

#include "stdint.h"

// Modbus RTU requests for reading NPK values
static uint8_t nitro[] = {0x01, 0x03, 0x00, 0x1e, 0x00, 0x01, 0xe4, 0x0c};
static uint8_t phos[]  = {0x01, 0x03, 0x00, 0x1f, 0x00, 0x01, 0xb5, 0xcc};
static uint8_t pota[]  = {0x01, 0x03, 0x00, 0x20, 0x00, 0x01, 0x85, 0xc0};

void UART2_Init(void);

void UART2_SendString(char *str);

void USART2_IRQHandler(void);

void UART2_SendData(uint8_t *data, uint8_t length);

void UART2_ReceiveData(uint8_t *data, uint8_t length);

void processElement(uint8_t *element, const char *elementName);

#endif /* SRC_UART_DATA_H_ */
```

UART_Data.c

```c
/*
 * UART_Data.c
 *
 *     This file provides the implementation for UART communication functions on
STM32F4xx microcontrollers.
 *     It includes UART2_Init for setting up UART2, UART2_SendString for
transmitting strings, and UART2_SendData and UART2_ReceiveData for handling
data transmission and reception.
 *     The USART2_IRQHandler function is used for handling USART2 interrupts,
primarily for data reception.
 *     Additionally, processElement function is provided for processing and
printing data received through UART.
 *     This module is crucial for projects involving serial communication, such
as interfacing with sensors or other peripherals.
 *
 *  Created on: Dec 17, 2023
 *      Author: Unmesh
 */

#include "stm32f4xx.h"
#include "Clock_config.h"
#include "UART_Data.h"
#include "stdio.h"

uint8_t Rx_Data[32];
uint8_t Rx_Index = 0;
uint8_t Tx_Data[8];

uint8_t Data_stored[10];
```

```
void UART2_Init(void) {
	RCC->AHB1ENR |= RCC_AHB1ENR_GPIODEN;   // Enable clocks for GPIOD and
USART2
	RCC->APB1ENR |= RCC_APB1ENR_USART2EN;

	GPIOD->MODER |= GPIO_MODER_MODER5_1 | GPIO_MODER_MODER6_1; // Set pins
PD5 and PD6 in alternate function mode for USART2
	GPIOD->AFR[0] |= (0x7 << 20) | (0x7 << 24); // Setting alternate function
7 (USART2) for PD5 and PD6

	USART2->BRR = 0x683;                                              //
Baud Rate Configuration
	USART2->CR1 |= USART_CR1_TE | USART_CR1_RE | USART_CR1_UE; // Enable
transmitter, receiver, and USART2 module

	USART2->CR1 |= USART_CR1_RXNEIE; // Enable interrupt on receive data
register
	NVIC_EnableIRQ(USART2_IRQn);                                      // Enable
USART2 interrupt
}

void UART2_SendString(char *str)         // Send string via UART2
{
	while (*str) {
		while (!(USART2->SR & USART_SR_TXE))
			;          // Wait until transmit data register is empty
		USART2->DR = *str++;
	}
}

void USART2_IRQHandler(void) {
    if (USART2->SR & USART_SR_RXNE) {    // If character received in USART2
        Rx_Data[Rx_Index++] = USART2->DR; // Read the character and store in
Rx_Data

        if (Rx_Index >= 32) {
            Rx_Index = 0; // Reset index if buffer is full
        }
    }
}

void UART2_SendData(uint8_t *data, uint8_t length) {
	// Set PD3 before sending data
	GPIOD->MODER |= GPIO_MODER_MODER3_0;  // Set PD3 as output
	GPIOD->BSRR = GPIO_BSRR_BS_3;         // Set PD3 high
	for (uint8_t i = 0; i < length; i++) {
		while (!(USART2->SR & USART_SR_TXE))
			; // Wait until transmit data register is empty
		USART2->DR = data[i];
	}
	while (!(USART2->SR & USART_SR_TC))
		; // Wait for transmission to complete

	// Reset PD3 after sending data
	GPIOD->BSRR = GPIO_BSRR_BR_3;         // Set PD3 low
}
```

```
void UART2_ReceiveData(uint8_t *data, uint8_t length) {
    for (uint8_t i = 0; i < length; i++) {
        if (i < Rx_Index) {
            data[i] = Rx_Data[i];
        } else {
            break;
        }
    }
    Rx_Index = 0;
}

void processElement(uint8_t *element, const char *elementName) {
    uint8_t response[32];

    UART2_SendData(element, sizeof(element));
    delay_ms(50);
    UART2_ReceiveData(response, sizeof(response));

    if (response[1] == 0x03) {
        uint16_t element_value = (response[3] << 8) | response[4];
        printf("%s Value: 0x%04X\n", elementName, element_value);
    }

    // Reset Rx_Index for next reading
    Rx_Index = 0;
}
```

8.4 Appendix - Data Sheets and Application Notes

The datasheets and application notes are included in the folder named “References” in the submission file.