SMART GREENHOUSE

BY GROUP 2

AKHIL R
BOOBATHI K

TABLE OF CONTENTS

PROJECT OVERVIEW

Introduction:

The SMART GREEN HOUSE project is designed to revolutionise the way we manage and control greenhouse environments. By integrating advanced microcontroller technology with modern cloud-based solutions, this project aims to create a sophisticated system for monitoring and managing greenhouse conditions with ease and precision.

In a traditional greenhouse, managing environmental factors such as temperature and humidity can be challenging, especially when aiming to optimise conditions for plant growth and overall productivity. The SMART GREEN HOUSE project addresses these challenges by utilising a combination of two STM32 F446RE microcontrollers and a Rugged Board A5D2X, orchestrated to provide a seamless and automated control system.

By leveraging these advanced technologies, the SMART GREEN HOUSE project represents a significant step forward in greenhouse automation, providing a powerful tool for optimising environmental conditions and enhancing agricultural productivity.

HARDWARE SPECIFICATIONS

1. System Components:

- **Rugged Board A5D2X (Master Controller)**
  - **Processor**: High-performance SOM with sufficient processing power for handling Modbus communication and control tasks.
  - **Communication Ports**: Multiple I/O ports for Modbus communication and connectivity with STM32 microcontrollers.
  - **Power Supply**: 12V DC or as per the Rugged Board specifications.
  - **Connectivity**: Support for network communication (e.g., Ethernet or Wi-Fi) for integration with PhyCloud and remote control.
  - **Operating System**: Compatible with the operating system or firmware for handling Modbus protocol and PhyCloud integration.
- **STM32 F446RE Microcontrollers (Slave Controllers)**
  - **Processor**: STM32F446RE microcontroller with ARM Cortex-M4 core.
  - **Clock Speed**: Up to 180 MHz.
  - **Flash Memory**: 512 KB.
  - **RAM**: 128 KB.
  - **Communication Interfaces**: UART, SPI, I2C, and GPIO for sensor data acquisition and Modbus communication.
  - **Analog-to-Digital Converter (ADC)**: For reading sensor data.
  - **Power Supply**: 3.3V or 5V DC as per STM32 specifications

- **Sensors and Actuators**

Temperature Sensor:

KY-013 analog temperature sensor, suitable for greenhouse environments.

- **Temperature Range:** -55°C to +125°C.
- **Accuracy:** ±0.5°C.
-
  - **Relay Module**: For controlling greenhouse devices (e.g., irrigation, fans, heaters).
    - **Relay Type**: SPST (Single Pole Single Throw) or as required.
    - **Relay Rating**: Suitable for the voltage and current requirements of the greenhouse devices.

2. Communication Protocol:

- **Modbus**: RTU or TCP for communication between the Rugged Board and STM32 microcontrollers.
    - **Baud Rate**: Configurable (commonly 9600, 19200, or 115200 bps).
    - **Data Bits**: 8.
    - **Parity**: None.
    - **Stop Bits**: 1 or 2.

3. Mobile App Integration:

- **PhyCloud Platform**: For remote monitoring and control.
    - **Connectivity**: Integration with PhyCloud through Internet connectivity (e.g., Wi-Fi, Ethernet).
    - **Mobile App**: Compatible with Android and iOS devices.
    - **Features**: Real-time data visualization, remote control, notifications.

4. Power Supply and Requirements:

- **Overall Power Supply**: Ensure a stable power source for the entire system.
- **Rugged Board**: Typically 12V DC or as specified by the manufacturer.
- **STM32 Microcontrollers**: Powered by 3.3V or 5V DC.
- **Power Consumption**: To be calculated based on the actual hardware configuration and connected devices.

5. Environmental Requirements:

- **Operating Temperature**: 0°C to 50°C (for indoor greenhouse environments).
- **Humidity**: Suitable for operation in a high-humidity environment.

6. Enclosures and Mounting:

- **Enclosures**: Weatherproof or protected enclosures for outdoor components to ensure durability.
- **Mounting**: Secure mounting for microcontrollers, sensors, and relays within the greenhouse.a

-

Modbus RTU (Remote Terminal Unit) Frame Structure:

- **Slave Address**: Identifies which slave device the master is communicating with. In this project, each STM32 slave has a unique Modbus address.
- **Function Code**: Indicates the operation to be performed. For example, reading sensor data (function code 03) or writing to a relay (function code 05).
- **Data**: Contains the payload of the message, such as the temperature reading or the command to control the relay.
- **CRC (Cyclic Redundancy Check)**: A checksum used for error checking to ensure data integrity during transmission.

Function Codes Used:

- **Read Holding Registers (Function Code 03)**: The master uses this function code to read the temperature data from Slave 2. The sensor data is typically stored in one or more holding registers.the function code 3 will be implemented on slave 2
- **Write Single Coil (Function Code 05)**: The master uses this function code to control the relay connected to Slave 1. This function writes a single bit to turn the relay on or off.The function code 5 is implemented on slave 1

7. Software and Firmware:

- **Firmware**: Custom firmware for STM32 microcontrollers for sensor data acquisition, Modbus communication, and relay control.
- **Software**: Configuration and control software for the Rugged Board, including simple Modbus management and PhyCloud integration.

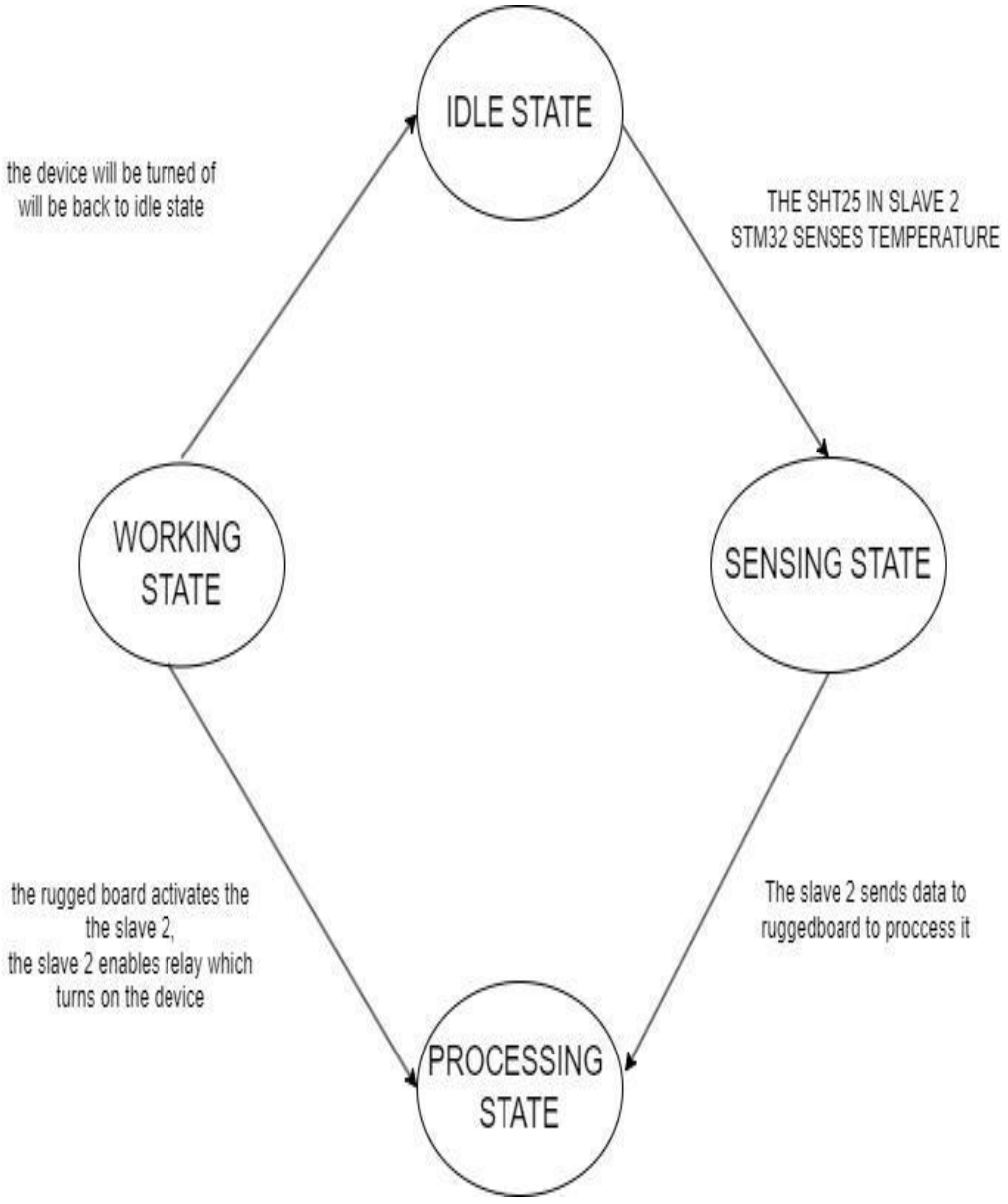
8. Safety and Compliance:

- **Standards**: Compliance with relevant electrical and safety standards.
- **Certifications**: Ensure components and system meet required certifications for safety and reliability.
- **9. Documentation:**

These specifications outline the key hardware and system requirements for the SMART GREEN HOUSE project, ensuring that all components work together effectively to achieve the desired automation and control capabilities.

BLOCK DIAGRAM

FINITE STATE MACHINE DIAGRAM



FINITE STATE MACHINE CODE:

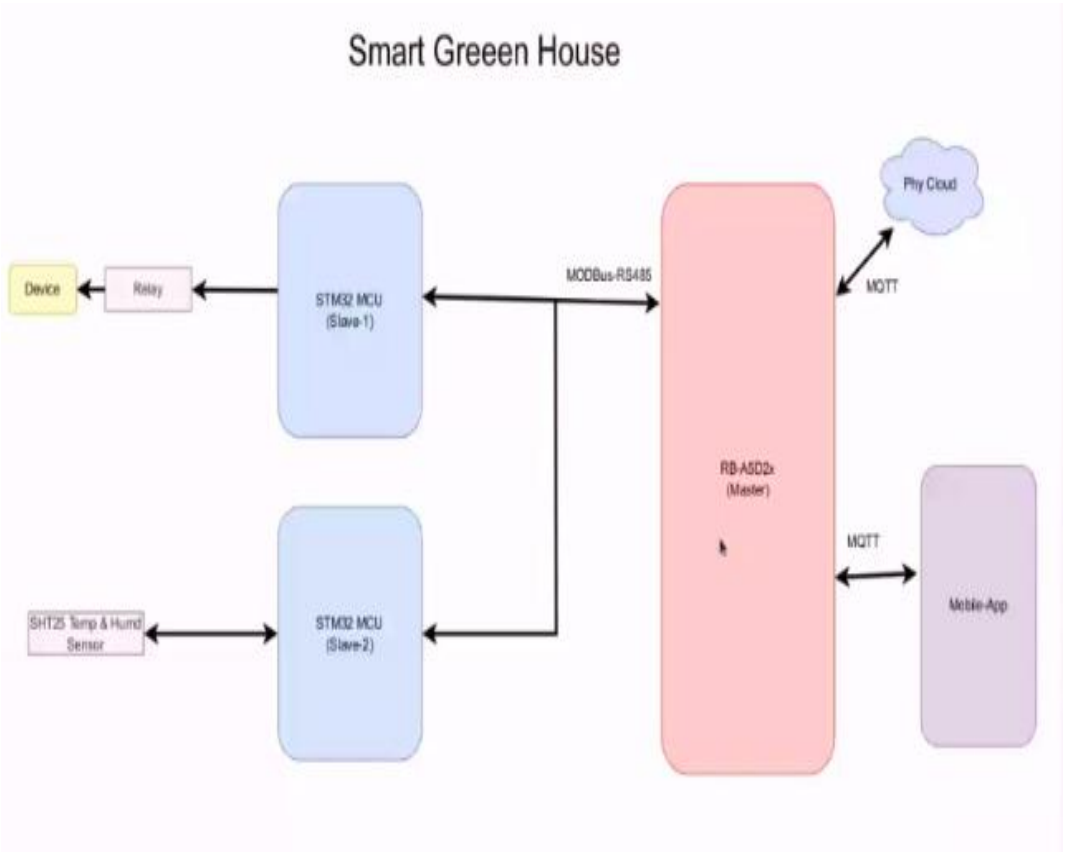
```
#include "modbus.h"  // Include your Modbus library
#include "mqtt.h"    // Include your MQTT library
#define CRITICAL_TEMP 25.0
#define READ_INTERVAL 10000 // 10 seconds

typedef enum {
   IDLE,
   READ_TEMP,
   CHECK_TEMP,
   CONTROL_RELAY,
   ERROR
} State;

State currentState = IDLE;
State nextState = IDLE;

float temperature = 0.0;
bool relayStatus = false;

void transitionTo(State state) {
   currentState = state;
```

```
switch (currentState) {
    case IDLE:
        // Code to handle IDLE state
        break;
    case READ_TEMP:
        // Code to read temperature from Slave 2
        temperature = readTemperatureFromSlave2();
        nextState = CHECK_TEMP;
        break;
    case CHECK_TEMP:
        // Code to check the temperature against the critical value
        if (temperature > CRITICAL_TEMP) {
            nextState = CONTROL_RELAY;
        } else {
            relayStatus = false;
            nextState = CONTROL_RELAY;
        }
        break;
    case CONTROL_RELAY:
        // Code to control the relay on Slave 1
        controlRelayOnSlave1(relayStatus);
        nextState = IDLE;
        break;
    case ERROR:
```

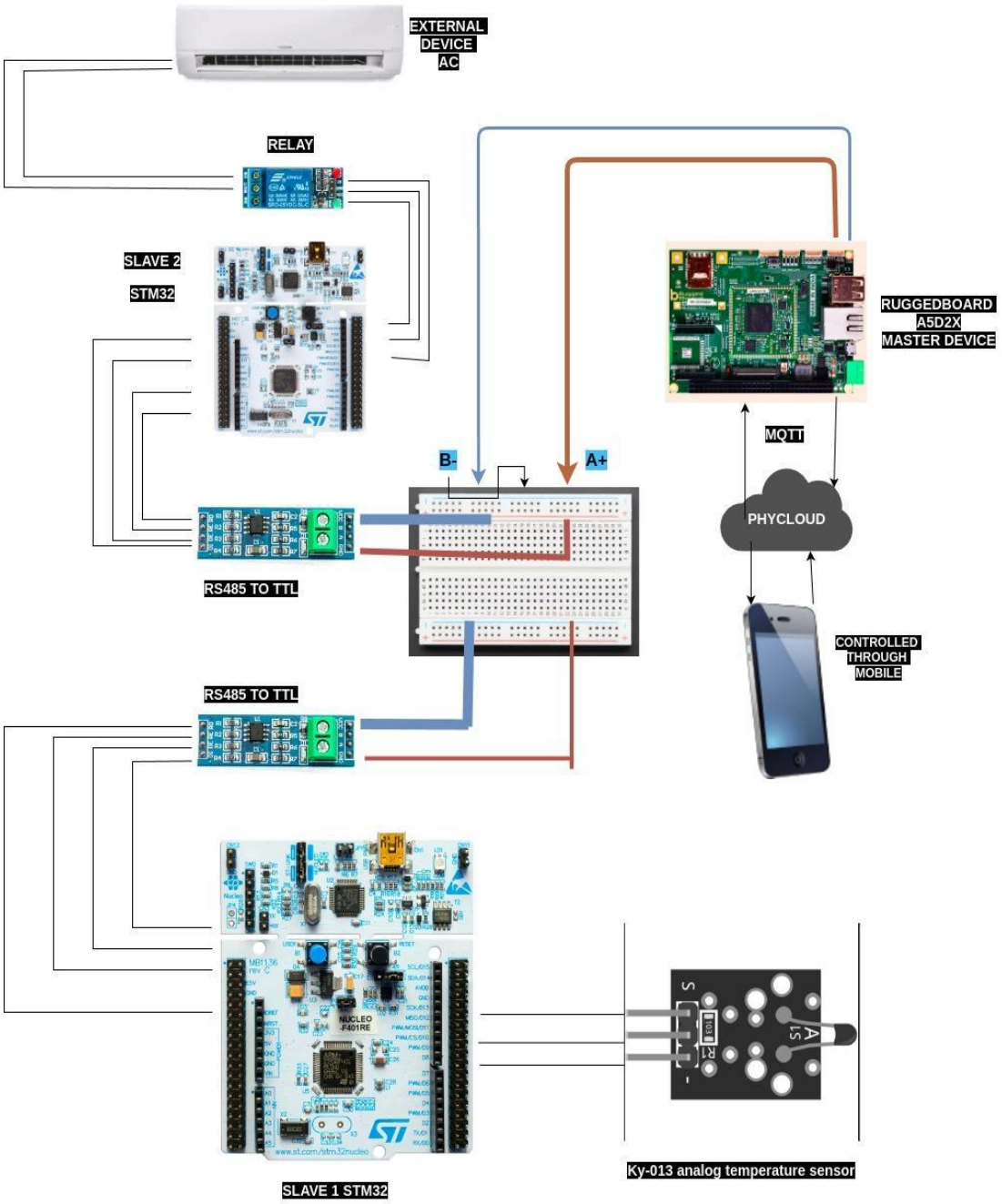
```
            // Handle errors
            break;
        }
}

float readTemperatureFromSlave2() {
    // Implementation to read temperature from Slave 2
    return 0.0;
}

void controlRelayOnSlave1(bool status) {
    // Implementation to control relay on Slave 1
}

void mainLoop() {
    while (1) {
        switch (currentState) {
            case IDLE:
                // Trigger state transition based on timer or event
                // e.g., every READ_INTERVAL seconds
                transitionTo(READ_TEMP);
                break;
            case READ_TEMP:
                // Transition already handled in the transitionTo function
```

```
            break;
        case CHECK_TEMP:
            // Transition already handled in the transitionTo function
            break;
        case CONTROL_RELAY:
            // Transition already handled in the transitionTo function
            break;
        case ERROR:
            // Handle error
            break;
    }
    // Wait for next cycle
    delay(READ_INTERVAL);
  }
}

int main() {
  // Initialization code
  initModbus();
  initMQTT();

  mainLoop();
  return 0;
}
```

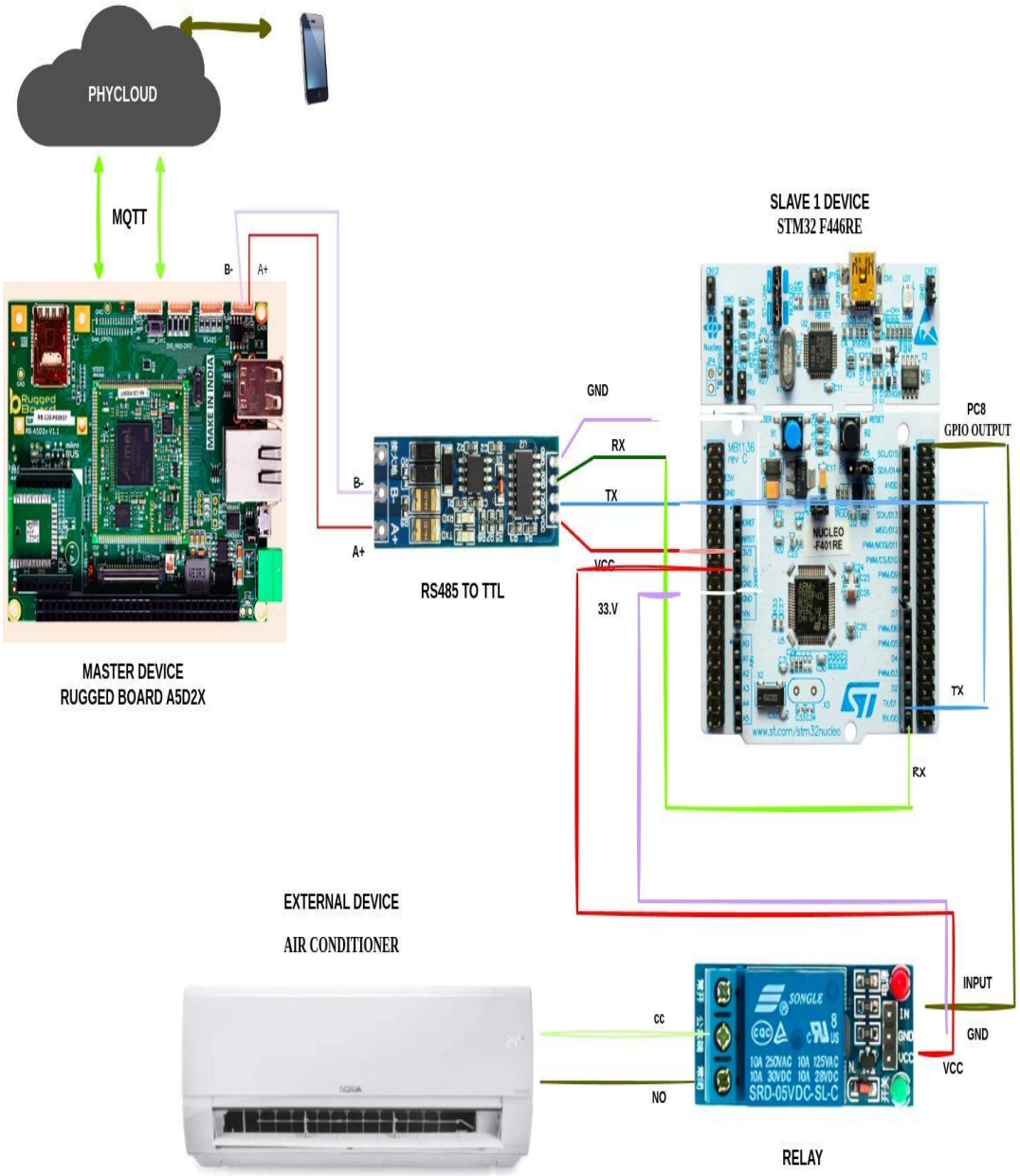
HIGH LEVEL BLOCK DIAGRAM



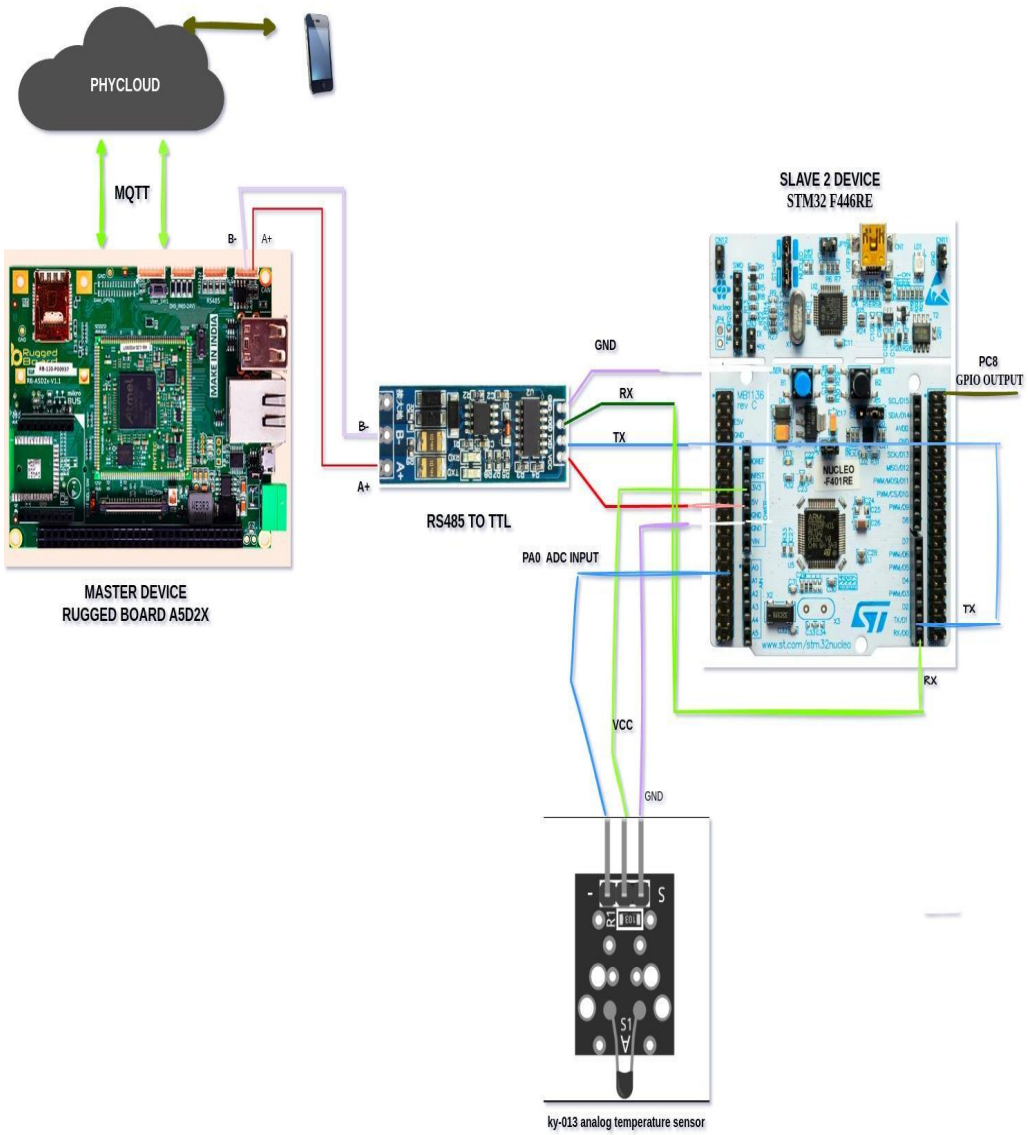
LOW LEVEL BLOCK DIAGRAM



CONNECTION DIAGRAM SLAVE 1



CONNECTION DIAGRAM SLAVE 2



TRANSITION STATE TABLE

CURRENT STATE	INPUT/ CONDITION	NEXT STATE	ACTION/ OUTPUT
IDLE	System start	**ky-013**	Initialise sensors and system components
SENSING State (ky-013)	Sensor data acquired	TRANSMIT DATA	Send temperature and humidity data to rugged board by slave 1 stm32
TRANSMIT DATA	Data Received by Rugged	PROCESS DATA	RUGGED BOARD PROCESS DATA RECEIVED BY SLAVE 2
PROCESS DATA	DATA processing complete	CONTROL DATA	Send command to Slave 2 to control relay
CONTROL RELAY	Relay activated	DEVICE ON	Turn on the connected green house device
DEVICE ON	Operation complete	IDLE	Return to idle state

Transitions:

- **Transition 1: From IDLE STATE to Sensing State when the system is triggered to sense environmental conditions (e.g., a timer event or external command).**

- **Transition 2: From Sensing State to Data Transmission State after successfully sensing the temperature and humidity values**

- **Transition 3: From Data Transmission State to Processing State when the Rugged Board A5D2X receives and acknowledges the data.**

- **Transition 4: From Processing State to Control State if the Rugged Board determines that an action is required (e.g., activating the relay).**

- **Transition 5: From Control State to Back to IDLE STATE after the relay is enabled, and the device is turned on.**

Peripheral Explanation

SYSTEM ON MODULE (RUGGED BOARD A5D2X)

- The Rugged Board A5D2X is a high-performance development platform centered around the Microchip SAMA5D2 ARM Cortex-A5 processor, making it ideal for industrial and IoT applications that demand reliable and robust performance. Designed for use in demanding environments, the board is built with industrial-grade construction and supports extended temperature ranges. It offers a comprehensive array of connectivity options, including Ethernet, USB, multiple UARTs, SPI, I2C, and CAN interfaces, along with support for SD cards, allowing for seamless integration with a wide range of peripherals and systems.

- The Rugged Board A5D2X is optimized for low-power consumption, making it suitable for energy-efficient and battery-powered applications. Its powerful processing capabilities and extensive I/O options make it a versatile platform for developing advanced embedded systems and resilient IoT solutions. Additionally, the board is equipped with advanced security features, including hardware encryption and secure boot, ensuring that sensitive data is protected and unauthorized access is prevented.

- The Rugged Board A5D2X is compatible with Linux and a variety of real-time operating systems, providing developers with access to a rich ecosystem of software and tools, which facilitates the rapid development and deployment of applications. Expansion headers on the board allow for the addition of custom modules, enabling tailored solutions for specific industrial or IoT needs. With its combination of durability, processing power, and flexibility, the Rugged Board A5D2X serves as the master device in this project, making it an excellent choice for mission-critical applications in fields such as automation, transportation, and remote monitoring.

STM32 F446RE (SLAVE 1)

- The STM32 F446RE microcontroller, based on the ARM Cortex-M4 core, is used in this project as Slave 1, responsible for controlling a relay connected to an external device. This microcontroller is well-suited for applications requiring real-time performance, low power consumption, and advanced peripherals. The STM32 F446RE features a range of connectivity options, including UART, SPI, and I2C, allowing it to communicate efficiently with the Rugged Board A5D2X via UART or RS-485.

- The relay control is managed through one of the STM32's GPIO pins, which is connected to a relay driver circuit. This circuit ensures that the relay is properly driven to control the external device based on the commands received from the Rugged Board. The STM32 F446RE's powerful processing capabilities and extensive peripheral support make it an excellent choice for handling the relay control tasks in this project.

STM32 F446RE (SLAVE 2)

- The STM32 F446RE microcontroller is also used as Slave 2 in this project, responsible for sensing the temperature using a KY-013 analog temperature sensor. This microcontroller's ARM Cortex-M4 core provides the necessary processing power to read the analog voltage output from the KY-013 sensor through its ADC (Analog-to-Digital Converter) pins. The sensor's output, which is proportional to the temperature, is continuously monitored by Slave 2.

- The STM32 F446RE communicates the temperature readings to the Rugged Board A5D2X via UART or RS-485. This data is then used by the Rugged Board to determine whether the temperature exceeds the critical threshold of 25°C, triggering a command to Slave 1 to control the relay accordingly. The STM32 F446RE's low power consumption and advanced peripheral set make it well-suited for temperature sensing applications in this project.

KY-013 TEMPERATURE SENSOR

- The KY-013 analog temperature sensor is used in this project to monitor the ambient temperature. The sensor outputs an analog voltage that varies with temperature, which is read by the ADC pins on the STM32 F446RE (Slave 2). The KY-013 sensor operates on a 5V power supply and provides a simple and effective means of temperature measurement. Its integration with the STM32 F446RE allows for real-time temperature monitoring and communication of this data to the Rugged Board A5D2X.

RELAY MODULE

- The relay module in this project is controlled by Slave 1 (STM32 F446RE) and is used to turn an external device on or off based on the temperature readings received from Slave 2. The relay is connected to one of the GPIO pins on the STM32, which controls the relay through a driver circuit. This setup ensures that the external device is activated when the temperature exceeds 25°C and deactivated when it falls below this threshold. The relay module is an essential component in managing the external device's operation, ensuring that it responds accurately to the temperature conditions monitored by the system.

INTERCONNECTIONS AND COMMUNICATION

- The Rugged Board A5D2X communicates with the two STM32 F446RE microcontrollers using UART or RS-485, providing a robust communication link between the master and slave devices. The Rugged Board serves as the master, issuing commands based on the temperature data received from Slave 2. Slave 1 controls the relay based on these commands, ensuring that the external device operates as intended.

- Power is supplied to all components, including the Rugged Board, STM32 microcontrollers, KY-013 sensor, and relay module, ensuring stable and reliable operation across the entire system. The Rugged Board's connection to the PhyCloud MQTT allows for remote monitoring and control, enhancing the project's capability to operate in an industrial or IoT environment.

- This hardware setup provides a comprehensive solution for temperature monitoring and control, leveraging the robust features of the Rugged Board A5D2X and STM32 microcontrollers to deliver a reliable and efficient system.

SLAVE 1 DEVICE CODE

Main.c

```
#include "main.h"

#include <stdio.h>

#include "modbusSlave.h"

ADC_HandleTypeDef hadc1;

TIM_HandleTypeDef htim6;

UART_HandleTypeDef huart1;

UART_HandleTypeDef huart2;

void SystemClock_Config(void);

static void MX_GPIO_Init(void);

static void MX_USART2_UART_Init(void);

static void MX_ADC1_Init(void);

static void MX_TIM6_Init(void);

static void MX_USART1_UART_Init(void);

uint8_t RxData[256];

uint8_t TxData[256];


uint16_t Holding_Registers_Data[50];




void HAL_UARTEx_RxEventCallback(UART_HandleTypeDef *huart, uint16_t Size)

{

        if (RxData[0] == SLAVE_ID)

        {

                switch (RxData[1]){
```

```
                case 0x03:
                        readHoldingRegs();
                        break;
                case 0x05:
                        writeSingleReg();
                        break;
                case 0x10:
                        writeHoldingRegs();
                        break;
                default:
                        modbusException(ILLEGAL_FUNCTION);
                        break;
                }
        }

        HAL_UARTEx_ReceiveToIdle_IT(&huart1, RxData, 256);
}
int main(void)
{
 HAL_Init();
 SystemClock_Config();

 MX_GPIO_Init();
 MX_USART2_UART_Init();
 MX_ADC1_Init();
 MX_TIM6_Init();
 MX_USART1_UART_Init();
          HAL_ADC_Start_IT(&hadc1);
   HAL_TIM_Base_Start_IT(&htim6);
```

```
    HAL_UARTEx_ReceiveToIdle_IT(&huart1, RxData, 256);
  while (1)
  {
  }
}
void SystemClock_Config(void)
{
  RCC_OscInitTypeDef RCC_OscInitStruct = {0};
  RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};

  __HAL_RCC_PWR_CLK_ENABLE();
  __HAL_PWR_VOLTAGESCALING_CONFIG(PWR_REGULATOR_VOLTAGE_SCALE3);
  * in the RCC_OscInitTypeDef structure.
  */
  RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSI;
  RCC_OscInitStruct.HSIState = RCC_HSI_ON;
  RCC_OscInitStruct.HSICalibrationValue = RCC_HSICALIBRATION_DEFAULT;
  RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
  RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSI;
  RCC_OscInitStruct.PLL.PLLM = 16;
  RCC_OscInitStruct.PLL.PLLN = 336;
  RCC_OscInitStruct.PLL.PLLP = RCC_PLLP_DIV4;
  RCC_OscInitStruct.PLL.PLLQ = 2;
  RCC_OscInitStruct.PLL.PLLR = 2;
  if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK)
  {
    Error_Handler();
  }
```

```
  RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK|RCC_CLOCKTYPE_SYSCLK
                              |RCC_CLOCKTYPE_PCLK1|RCC_CLOCKTYPE_PCLK2;
  RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK;
  RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
  RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV2;
  RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV1;

  if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_2) != HAL_OK)
  {
    Error_Handler();
  }
}


  ADC_ChannelConfTypeDef sConfig = {0};

  /* USER CODE BEGIN ADC1_Init 1 */

  /* USER CODE END ADC1_Init 1 */

  /** Configure the global features of the ADC (Clock, Resolution, Data Alignment and number of conversion)
  */
  hadc1.Instance = ADC1;
  hadc1.Init.ClockPrescaler = ADC_CLOCK_SYNC_PCLK_DIV4;
  hadc1.Init.Resolution = ADC_RESOLUTION_12B;
  hadc1.Init.ScanConvMode = DISABLE;
  hadc1.Init.ContinuousConvMode = ENABLE;
  hadc1.Init.DiscontinuousConvMode = DISABLE;
  hadc1.Init.ExternalTrigConvEdge = ADC_EXTERNALTRIGCONVEDGE_NONE;
```

```
  hadc1.Init.ExternalTrigConv = ADC_SOFTWARE_START;
  hadc1.Init.DataAlign = ADC_DATAALIGN_RIGHT;
  hadc1.Init.NbrOfConversion = 1;
  hadc1.Init.DMAContinuousRequests = DISABLE;
  hadc1.Init.EOCSelection = ADC_EOC_SINGLE_CONV;
  if (HAL_ADC_Init(&hadc1) != HAL_OK)
  {
    Error_Handler();
  }
  sConfig.Channel = ADC_CHANNEL_0;
  sConfig.Rank = 1;
  sConfig.SamplingTime = ADC_SAMPLETIME_3CYCLES;
  if (HAL_ADC_ConfigChannel(&hadc1, &sConfig) != HAL_OK)
  {
    Error_Handler();
  }
}
static void MX_TIM6_Init(void)
{
  TIM_MasterConfigTypeDef sMasterConfig = {0};

  /* USER CODE BEGIN TIM6_Init 1 */

  /* USER CODE END TIM6_Init 1 */
  htim6.Instance = TIM6;
  htim6.Init.Prescaler = 8399;
  htim6.Init.CounterMode = TIM_COUNTERMODE_UP;
  htim6.Init.Period = 9999;
  htim6.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_DISABLE;
```

```
  if (HAL_TIM_Base_Init(&htim6) != HAL_OK)
  {
    Error_Handler();
  }
  sMasterConfig.MasterOutputTrigger = TIM_TRGO_RESET;
  sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
  if (HAL_TIMEx_MasterConfigSynchronization(&htim6, &sMasterConfig) != HAL_OK)
  {
    Error_Handler();
  }
  /* USER CODE BEGIN TIM6_Init 2 */


  /* USER CODE END TIM6_Init 2 */


}


/**
  * @brief USART1 Initialization Function
  * @param None
  * @retval None
  */
static void MX_USART1_UART_Init(void)
{


   huart1.Instance = USART1;
  huart1.Init.BaudRate = 9600;
  huart1.Init.WordLength = UART_WORDLENGTH_8B;
  huart1.Init.StopBits = UART_STOPBITS_1;
  huart1.Init.Parity = UART_PARITY_NONE;
```

```
  huart1.Init.Mode = UART_MODE_TX_RX;
  huart1.Init.HwFlowCtl = UART_HWCONTROL_NONE;
  huart1.Init.OverSampling = UART_OVERSAMPLING_16;
  if (HAL_UART_Init(&huart1) != HAL_OK)
  {
    Error_Handler();
}

/**
  * @brief USART2 Initialization Function
  * @param None
  * @retval None
  */
static void MX_USART2_UART_Init(void)
{

  /* USER CODE BEGIN USART2_Init 0 */

  /* USER CODE END USART2_Init 0 */

  /* USER CODE BEGIN USART2_Init 1 */

  /* USER CODE END USART2_Init 1 */
  huart2.Instance = USART2;
  huart2.Init.BaudRate = 115200;
  huart2.Init.WordLength = UART_WORDLENGTH_8B;
  huart2.Init.StopBits = UART_STOPBITS_1;
  huart2.Init.Parity = UART_PARITY_NONE;
  huart2.Init.Mode = UART_MODE_TX_RX;
```

```
  huart2.Init.HwFlowCtl = UART_HWCONTROL_NONE;
  huart2.Init.OverSampling = UART_OVERSAMPLING_16;
  if (HAL_UART_Init(&huart2) != HAL_OK)
  {
    Error_Handler();
  }
  /* USER CODE BEGIN USART2_Init 2 */


  /* USER CODE END USART2_Init 2 */


}


/**
  * @brief GPIO Initialization Function
  * @param None
  * @retval None
  */
static void MX_GPIO_Init(void)
{
  GPIO_InitTypeDef GPIO_InitStruct = {0};
/* USER CODE BEGIN MX_GPIO_Init_1 */
/* USER CODE END MX_GPIO_Init_1 */


  /* GPIO Ports Clock Enable */
  __HAL_RCC_GPIOC_CLK_ENABLE();
  __HAL_RCC_GPIOH_CLK_ENABLE();
  __HAL_RCC_GPIOA_CLK_ENABLE();
  __HAL_RCC_GPIOB_CLK_ENABLE();
```

```
  /*Configure GPIO pin Output Level */
  HAL_GPIO_WritePin(LD2_GPIO_Port, LD2_Pin, GPIO_PIN_RESET);

  /*Configure GPIO pin : B1_Pin */
  GPIO_InitStruct.Pin = B1_Pin;
  GPIO_InitStruct.Mode = GPIO_MODE_IT_FALLING;
  GPIO_InitStruct.Pull = GPIO_NOPULL;
  HAL_GPIO_Init(B1_GPIO_Port, &GPIO_InitStruct);

  /*Configure GPIO pin : LD2_Pin */
  GPIO_InitStruct.Pin = LD2_Pin;
  GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
  GPIO_InitStruct.Pull = GPIO_NOPULL;
  GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
  HAL_GPIO_Init(LD2_GPIO_Port, &GPIO_InitStruct);

/* USER CODE BEGIN MX_GPIO_Init_2 */
/* USER CODE END MX_GPIO_Init_2 */
}

/* USER CODE BEGIN 4 */


void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim) {
    if (htim->Instance == TIM6) {
        /* Start ADC conversion on timer interrupt */
        HAL_ADC_Start_IT(&hadc1);
    }
}
```

```
/* USER CODE END 4 */


/**
  * @brief  This function is executed in case of error occurrence.
  * @retval None
  */
void Error_Handler(void)
{
  /* USER CODE BEGIN Error_Handler_Debug */
  /* User can add his own implementation to report the HAL error return state */
  __disable_irq();
  while (1)
  {
  }
  /* USER CODE END Error_Handler_Debug */
}


#ifdef  USE_FULL_ASSERT
  */
void assert_failed(uint8_t *file, uint32_t line)
{
#endif /* USE_FULL_ASSERT */
```

Main.h

```
#ifndef __MAIN_H

#define __MAIN_H


#ifdef __cplusplus

extern "C" {

#endif


#include "stm32f4xx_hal.h"


void Error_Handler(void);

#define B1_Pin GPIO_PIN_13

#define B1_GPIO_Port GPIOC

#define USART_TX_Pin GPIO_PIN_2

#define USART_TX_GPIO_Port GPIOA

#define USART_RX_Pin GPIO_PIN_3

#define USART_RX_GPIO_Port GPIOA

#define LD2_Pin GPIO_PIN_5

#define LD2_GPIO_Port GPIOA

#define TMS_Pin GPIO_PIN_13

#define TMS_GPIO_Port GPIOA

#define TCK_Pin GPIO_PIN_14

#define TCK_GPIO_Port GPIOA

#define SWO_Pin GPIO_PIN_3

#define SWO_GPIO_Port GPIOB
```

```
/* USER CODE BEGIN Private defines */

/* USER CODE END Private defines */

#ifdef __cplusplus
}
#endif

#endif
```

Modbus_crc.c

```
#include "stdint.h"

static const uint8_t table_crc_hi[] = {
    0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x01, 0xC0,
    0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41,
    0x00, 0xC1, 0x81, 0x40, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0,
    0x80, 0x41, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40,
    0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1,
    0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x01, 0xC0, 0x80, 0x41,
    0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1,
    0x81, 0x40, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41,
    0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x01, 0xC0,
```

```
    0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x00, 0xC1, 0x81, 0x40,
    0x01, 0xC0, 0x80, 0x41, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1,
    0x81, 0x40, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41,
    0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x01, 0xC0,
    0x80, 0x41, 0x00, 0xC1, 0x81, 0x40
};


static const uint8_t table_crc_lo[] = {
    0x00, 0xC0, 0xC1, 0x01, 0xC3, 0x03, 0x02, 0xC2, 0xC6, 0x06,
    0x07, 0xC7, 0x05, 0xC5, 0xC4, 0x04, 0xCC, 0x0C, 0x0D, 0xCD,
    0x0F, 0xCF, 0xCE, 0x0E, 0x0A, 0xCA, 0xCB, 0x0B, 0xC9, 0x09,
    0x08, 0xC8, 0xD8, 0x18, 0x19, 0xD9, 0x1B, 0xDB, 0xDA, 0x1A,
    0x1E, 0xDE, 0xDF, 0x1F, 0xDD, 0x1D, 0x1C, 0xDC, 0x14, 0xD4,
    0xD5, 0x15, 0xD7, 0x17, 0x16, 0xD6, 0xD2, 0x12, 0x13, 0xD3,
    0x11, 0xD1, 0xD0, 0x10, 0xF0, 0x30, 0x31, 0xF1, 0x33, 0xF3,
    0xF2, 0x32, 0x36, 0xF6, 0xF7, 0x37, 0xF5, 0x35, 0x34, 0xF4,
    0x3C, 0xFC, 0xFD, 0x3D, 0xFF, 0x3F, 0x3E, 0xFE, 0xFA, 0x3A,
    0x3B, 0xFB, 0x39, 0xF9, 0xF8, 0x38, 0x28, 0xE8, 0xE9, 0x29,
    0xEB, 0x2B, 0x2A, 0xEA, 0xEE, 0x2E, 0x2F, 0xEF, 0x2D, 0xED,
    0xEC, 0x2C, 0xE4, 0x24, 0x25, 0xE5, 0x27, 0xE7, 0xE6, 0x26,
    0x22, 0xE2, 0xE3, 0x23, 0xE1, 0x21, 0x20, 0xE0, 0xA0, 0x60,
    0x61, 0xA1, 0x63, 0xA3, 0xA2, 0x62, 0x66, 0xA6, 0xA7, 0x67,
    0xA5, 0x65, 0x64, 0xA4, 0x6C, 0xAC, 0xAD, 0x6D, 0xAF, 0x6F,
    0x6E, 0xAE, 0xAA, 0x6A, 0x6B, 0xAB, 0x69, 0xA9, 0xA8, 0x68,
    0x78, 0xB8, 0xB9, 0x79, 0xBB, 0x7B, 0x7A, 0xBA, 0xBE, 0x7E,
```

```
    0x7F, 0xBF, 0x7D, 0xBD, 0xBC, 0x7C, 0xB4, 0x74, 0x75, 0xB5,
    0x77, 0xB7, 0xB6, 0x76, 0x72, 0xB2, 0xB3, 0x73, 0xB1, 0x71,
    0x70, 0xB0, 0x50, 0x90, 0x91, 0x51, 0x93, 0x53, 0x52, 0x92,
    0x96, 0x56, 0x57, 0x97, 0x55, 0x95, 0x94, 0x54, 0x9C, 0x5C,
    0x5D, 0x9D, 0x5F, 0x9F, 0x9E, 0x5E, 0x5A, 0x9A, 0x9B, 0x5B,
    0x99, 0x59, 0x58, 0x98, 0x88, 0x48, 0x49, 0x89, 0x4B, 0x8B,
    0x8A, 0x4A, 0x4E, 0x8E, 0x8F, 0x4F, 0x8D, 0x4D, 0x4C, 0x8C,
    0x44, 0x84, 0x85, 0x45, 0x87, 0x47, 0x46, 0x86, 0x82, 0x42,
    0x43, 0x83, 0x41, 0x81, 0x80, 0x40
};


uint16_t crc16(uint8_t *buffer, uint16_t buffer_length)
{
    uint8_t crc_hi = 0xFF;
    uint8_t crc_lo = 0xFF;
    unsigned int i;


    while (buffer_length--) {
        i = crc_lo ^ *buffer++;
        crc_lo = crc_hi ^ table_crc_hi[i];
        crc_hi = table_crc_lo[i];
    }


    return (crc_hi << 8 | crc_lo);
}
```

Modbus_crc.h

```
#ifndef INC_MODBUS_CRC_H_
#define INC_MODBUS_CRC_H_
#include "stdint.h"
uint16_t crc16(uint8_t *buffer, uint16_t buffer_length)
#endif
```

ModbusSlave.c

```
#include "modbusSlave.h"
#include "string.h"
#include "stm32f4xx_hal.h"
#include "stdio.h"

extern uint8_t RxData[256];
extern uint8_t TxData[256];
extern uint16_t Holding_Registers_Data[50];
extern UART_HandleTypeDef huart1;

extern ADC_HandleTypeDef hadc1;
```

```
extern uint32_t adcValue;
uint16_t pot_value;


void HAL_ADC_ConvCpltCallback(ADC_HandleTypeDef* hadc) {
    if (hadc->Instance == ADC1) {
        uint32_t adcValue = HAL_ADC_GetValue(hadc);
        pot_value = (adcValue / 80);
        printf("Temp:  %d\n", pot_value);
        writeHoldingRegs();
    }
}


void sendData(uint8_t *data, int size) {
    uint16_t crc = crc16(data, size);
    data[size] = crc & 0xFF;
    data[size + 1] = (crc >> 8) & 0xFF;
    HAL_UART_Transmit(&huart1, data, size + 2, 1000);
}


void modbusException(uint8_t exceptionCode) {
    TxData[0] = RxData[0];
    TxData[1] = RxData[1] | 0x80;
    TxData[2] = exceptionCode;
    sendData(TxData, 3);
}
```

```
uint8_t writeHoldingRegs(void) {
  uint16_t startAddr = ((RxData[2] << 8) | RxData[3]);
  uint16_t numRegs = ((RxData[4] << 8) | RxData[5]);

  if (numRegs < 1 || numRegs > 123) {
    modbusException(ILLEGAL_DATA_VALUE);
    return 0;
  }

  uint16_t endAddr = startAddr + numRegs - 1;
  if (endAddr > 49) {
    modbusException(ILLEGAL_DATA_ADDRESS);
    return 0;
  }

  int indx = 7;
  for (int i = 0; i < numRegs; i++) {
    if (startAddr == 10) {
      Holding_Registers_Data[startAddr] = pot_value;
    } else {
      Holding_Registers_Data[startAddr] = (RxData[indx] << 8) | RxData[indx + 1];
    }
    startAddr++;
    indx += 2;
```

```
    }

    TxData[0] = SLAVE_ID;
    TxData[1] = RxData[1];
    TxData[2] = RxData[2];
    TxData[3] = RxData[3];
    TxData[4] = RxData[4];
    TxData[5] = RxData[5];

    sendData(TxData, 6);
    return 1;
}

uint8_t readHoldingRegs(void) {
    uint16_t startAddr = ((RxData[2] << 8) | RxData[3]);
    uint16_t numRegs = ((RxData[4] << 8) | RxData[5]);

    if (numRegs < 1 || numRegs > 125) {
        modbusException(ILLEGAL_DATA_VALUE);
        return 0;
    }

    uint16_t endAddr = startAddr + numRegs - 1;
    if (endAddr > 49) {
        modbusException(ILLEGAL_DATA_ADDRESS);
```

```
        return 0;
    }

    TxData[0] = SLAVE_ID;
    TxData[1] = RxData[1];
    TxData[2] = numRegs * 2;
    int indx = 3;

    for (int i = 0; i < numRegs; i++) {
        uint16_t regValue = 0;
        if (startAddr == 10) {
            regValue = pot_value;
        } else {
            regValue = Holding_Registers_Data[startAddr];
        }

        TxData[indx++] = (regValue >> 8) & 0xFF;
        TxData[indx++] = regValue & 0xFF;
        startAddr++;
    }

    sendData(TxData, indx);
    return 1;
}
```

```
uint8_t writeSingleReg(void) {
   uint16_t startAddr = ((RxData[2] << 8) | RxData[3]);

   if (startAddr > 49) {
      modbusException(ILLEGAL_DATA_ADDRESS);
      return 0;
   }

   Holding_Registers_Data[startAddr] = (RxData[4] << 8) | RxData[5];

   TxData[0] = SLAVE_ID;
   TxData[1] = RxData[1];
   TxData[2] = RxData[2];
   TxData[3] = RxData[3];
   TxData[4] = RxData[4];
   TxData[5] = RxData[5];

   sendData(TxData, 6);
   return 1;
}
```

ModbusSlave.h

```
 #ifndef INC_MODBUSSLAVE_H_
#define INC_MODBUSSLAVE_H_
```

```
#include "modbus_crc.h"

#include "stm32f4xx_hal.h"



#define SLAVE_ID 0x02



#define ILLEGAL_FUNCTION       0x01

#define ILLEGAL_DATA_ADDRESS   0x02

#define ILLEGAL_DATA_VALUE     0x03




static uint16_t Holding_Registers_Database[50]={

                0000, 1111, 2222, 3333, 4444, 5555, 6666, 7777, 8888, 9999,   // 0-9 40001-40010

                12345, 15432, 15535, 10234, 19876, 13579, 10293, 19827, 13456, 14567,  // 10-19 40011-40020

                21345, 22345, 24567, 25678, 26789, 24680, 20394, 29384, 26937, 27654,  // 20-29 40021-40030

                31245, 31456, 34567, 35678, 36789, 37890, 30948, 34958, 35867, 36092,  // 30-39 40031-40040

                45678, 46789, 47890, 41235, 42356, 43567, 40596, 49586, 48765, 41029,  // 40-49 40041-40050

};



static const uint16_t Input_Registers_Database[50]={

                0000, 1111, 2222, 3333, 4444, 5555, 6666, 7777, 8888, 9999,   // 0-9 30001-30010

                12345, 15432, 15535, 10234, 19876, 13579, 10293, 19827, 13456, 14567,  // 10-19 30011-30020
```

```
                21345, 22345, 24567, 25678, 26789, 24680, 20394, 29384, 26937, 27654,  //
20-29 30021-30030

                31245, 31456, 34567, 35678, 36789, 37890, 30948, 34958, 35867, 36092,  //
30-39 30031-30040

                45678, 46789, 47890, 41235, 42356, 43567, 40596, 49586, 48765, 41029,  //
40-49 30041-30050

};



static uint8_t Coils_Database[25]={

                0b01001001, 0b10011100, 0b10101010, 0b01010101, 0b11001100,     // 0-39
1-40

                0b10100011, 0b01100110, 0b10101111, 0b01100000, 0b10111100,    // 40-79
41-80

                0b11001100, 0b01101100, 0b01010011, 0b11111111, 0b00000000,     // 80-
119  81-120

                0b01010101, 0b00111100, 0b00001111, 0b11110000, 0b10001111,     // 120-
159 121-160

                0b01010100, 0b10011001, 0b11111000, 0b00001101, 0b00101010,     // 160-
199 161-200

};



static const uint8_t Inputs_Database[25]={

                0b01001001, 0b10011100, 0b10101010, 0b01010101, 0b11001100,     // 0-39
10001-10040

                0b10100011, 0b01100110, 0b10101111, 0b01100000, 0b10111100,    // 40-79
10041-10080

                0b11001100, 0b01101100, 0b01010011, 0b11111111, 0b00000000,     // 80-
119  10081-10120

                0b01010101, 0b00111100, 0b00001111, 0b11110000, 0b10001111,     // 120-
159 10121-10160
```

```
                0b01010100, 0b10011001, 0b11111000, 0b00001101, 0b00101010,    // 160-199 10161-10200

};



uint8_t readHoldingRegs (void);

uint8_t readInputRegs (void);


uint8_t readCoils (void);

uint8_t readInputs (void);


uint8_t writeSingleReg (void);

uint8_t writeHoldingRegs (void);


void modbusException (uint8_t exceptioncode);

void HAL_ADC_ConvCpltCallback(ADC_HandleTypeDef* hadc);

#endif  /* INC_MODBUSSLAVE_H_ */
```

SLAVE 2 DEVICE CODE

Main.c

```
#include "main.h"
#include "modbusSlave.h"

UART_HandleTypeDef huart2;

uint8_t RxData[256];
uint8_t TxData[256];

void HAL_UARTEx_RxEventCallback(UART_HandleTypeDef *huart, uint16_t Size)
{
  if (RxData[0] == SLAVE_ID)
  {
    switch (RxData[1]){
    case 0x03:
      readHoldingRegs();
      break;
    case 0x05:
      writeSingleReg();
      break;
    case 0x10:
```

```
            writeHoldingRegs();
            break;
        default:
            modbusException(ILLEGAL_FUNCTION);
            break;
        }
    }

    HAL_UARTEx_ReceiveToIdle_IT(&huart2, RxData, 256);
}

int main(void)
{
    HAL_Init();
    SystemClock_Config();
    MX_GPIO_Init();
    MX_USART2_UART_Init();

    HAL_UARTEx_ReceiveToIdle_IT(&huart2, RxData, 256);

    while (1)
    {
    }
}
```

```
void SystemClock_Config(void)
{
  RCC_OscInitTypeDef RCC_OscInitStruct = {0};
  RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};

  __HAL_RCC_PWR_CLK_ENABLE();

__HAL_PWR_VOLTAGESCALING_CONFIG(PWR_REGULATOR_VOLTAGE_SCALE
1);

  RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSE;
  RCC_OscInitStruct.HSEState = RCC_HSE_ON;
  RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
  RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSE;
  RCC_OscInitStruct.PLL.PLLM = 4;
  RCC_OscInitStruct.PLL.PLLN = 180;
  RCC_OscInitStruct.PLL.PLLP = RCC_PLLP_DIV2;
  RCC_OscInitStruct.PLL.PLLQ = 2;
  RCC_OscInitStruct.PLL.PLLR = 2;
  if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK)
  {
    Error_Handler();
  }

  if (HAL_PWREx_EnableOverDrive() != HAL_OK)
  {
```

```
    Error_Handler();
  }

  RCC_ClkInitStruct.ClockType                                                    =
RCC_CLOCKTYPE_HCLK|RCC_CLOCKTYPE_SYSCLK
                        |RCC_CLOCKTYPE_PCLK1|RCC_CLOCKTYPE_PCLK2;
  RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK;
  RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
  RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV4;
  RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV2;

  if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_5) != HAL_OK)
  {
    Error_Handler();
  }
}

static void MX_USART2_UART_Init(void)
{
  huart2.Instance = USART2;
  huart2.Init.BaudRate = 19200;
  huart2.Init.WordLength = UART_WORDLENGTH_8B;
  huart2.Init.StopBits = UART_STOPBITS_1;
  huart2.Init.Parity = UART_PARITY_NONE;
  huart2.Init.Mode = UART_MODE_TX_RX;
```

```
  huart2.Init.HwFlowCtl = UART_HWCONTROL_NONE;
  huart2.Init.OverSampling = UART_OVERSAMPLING_16;
  if (HAL_UART_Init(&huart2) != HAL_OK)
  {
    Error_Handler();
  }
}

static void MX_GPIO_Init(void)
{
  GPIO_InitTypeDef GPIO_InitStruct = {0};

  __HAL_RCC_GPIOH_CLK_ENABLE();
  __HAL_RCC_GPIOA_CLK_ENABLE();
  __HAL_RCC_GPIOC_CLK_ENABLE();

  HAL_GPIO_WritePin(Relay_out_GPIO_Port, Relay_out_Pin, GPIO_PIN_RESET);

  GPIO_InitStruct.Pin = Relay_out_Pin;
  GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
  GPIO_InitStruct.Pull = GPIO_NOPULL;
  GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
  HAL_GPIO_Init(Relay_out_GPIO_Port, &GPIO_InitStruct);
}
```

```
void Error_Handler(void)
{
  __disable_irq();
  while (1)
  {
  }
}


#ifdef  USE_FULL_ASSERT
void assert_failed(uint8_t *file, uint32_t line)
{
}
#endif /* USE_FULL_ASSERT */
```

Main.h

```
#ifndef __MAIN_H
#define __MAIN_H


#ifdef __cplusplus
extern "C" {
#endif


#include "stm32f4xx_hal.h"


void Error_Handler(void);
```

```
#define Relay_out_Pin GPIO_PIN_8
#define Relay_out_GPIO_Port GPIOC

#ifdef __cplusplus
}
#endif

#endif /* __MAIN_H */
```

ModbusSlave.c

```
#include "modbusSlave.h"
#include "string.h"
#include "main.h"
extern uint8_t RxData[256];
extern uint8_t TxData[256];
extern UART_HandleTypeDef huart2;

#define RELAY_REGISTER_ADDRESS 0x05



void turnRelayOn(void)
{
```

```
   HAL_GPIO_WritePin(Relay_out_GPIO_Port,  Relay_out_Pin,  GPIO_PIN_SET);      // Replace GPIOx and GPIO_PIN_y with actual GPIO port and pin

}


// Function to turn the relay off

void turnRelayOff(void)

{

   HAL_GPIO_WritePin(Relay_out_GPIO_Port,  Relay_out_Pin,  GPIO_PIN_RESET);    // Replace GPIOx and GPIO_PIN_y with actual GPIO port and pin

}


void sendData (uint8_t *data, int size)

{

        // we will calculate the CRC in this function itself

        uint16_t crc = crc16(data, size);

        data[size] = crc&0xFF;   // CRC LOW

        data[size+1] = (crc>>8)&0xFF;  // CRC HIGH


        HAL_UART_Transmit(&huart2, data, size+2, 1000);

}


void modbusException (uint8_t exceptioncode)

{

        //| SLAVE_ID | FUNCTION_CODE | Exception code | CRC     |

        //| 1 BYTE   |  1 BYTE       |    1 BYTE      | 2 BYTES |
```

```
	TxData[0] = RxData[0];	// slave ID
	TxData[1] = RxData[1]|0x80;  // adding 1 to the MSB of the function code
	TxData[2] = exceptioncode;   // Load the Exception code
	sendData(TxData, 3);	// send Data... CRC will be calculated in the function
}



uint8_t readHoldingRegs (void)
{
	uint16_t startAddr = ((RxData[2]<<8)|RxData[3]);  // start Register Address

	uint16_t numRegs = ((RxData[4]<<8)|RxData[5]);    // number to registers master has requested
	if ((numRegs<1)||(numRegs>125))  // maximum no. of Registers as per the PDF
	{
		modbusException (ILLEGAL_DATA_VALUE);  // send an exception
		return 0;
	}

	uint16_t endAddr = startAddr+numRegs-1;  // end Register
	if (endAddr>49)  // end Register can not be more than 49 as we only have record of 50 Registers in total
	{
		modbusException(ILLEGAL_DATA_ADDRESS);   // send an exception
		return 0;
	}
```

```
	// Prepare TxData buffer

	//| SLAVE_ID | FUNCTION_CODE | BYTE COUNT | DATA      | CRC     |
	//| 1 BYTE   |  1 BYTE       |  1 BYTE    | N*2 BYTES | 2 BYTES |

	TxData[0] = SLAVE_ID;  // slave ID
	TxData[1] = RxData[1];  // function code
	TxData[2] = numRegs*2;  // Byte count
	int indx = 3;  // we need to keep track of how many bytes has been stored in TxData Buffer

	for (int i=0; i<numRegs; i++)   // Load the actual data into TxData buffer
	{
		TxData[indx++]  =  (Holding_Registers_Database[startAddr]>>8)&0xFF;    // extract the higher byte
		TxData[indx++]  =  (Holding_Registers_Database[startAddr])&0xFF;       // extract the lower byte
		startAddr++;  // increment the register address
	}

	sendData(TxData, indx);  // send data... CRC will be calculated in the function itself
	return 1;   // success
}
```

```
uint8_t writeHoldingRegs (void)
{
	uint16_t startAddr = ((RxData[2]<<8)|RxData[3]);  // start Register Address

	uint16_t numRegs = ((RxData[4]<<8)|RxData[5]);    // number to registers master has requested
	if ((numRegs<1)||(numRegs>123))  // maximum no. of Registers as per the PDF
	{
		modbusException (ILLEGAL_DATA_VALUE);  // send an exception
		return 0;
	}

	uint16_t endAddr = startAddr+numRegs-1;  // end Register
	if (endAddr>49)  // end Register can not be more than 49 as we only have record of 50 Registers in total
	{
		modbusException(ILLEGAL_DATA_ADDRESS);   // send an exception
		return 0;
	}

	/* start saving 16 bit data
	 * Data starts from RxData[7] and we need to combine 2 bytes together
	 * 16 bit Data = firstByte<<8|secondByte
	 */
	int indx = 7;  // we need to keep track of index in RxData
	for (int i=0; i<numRegs; i++)
```

```
        {
                Holding_Registers_Database[startAddr++] = (RxData[indx++]<<8)|RxData[indx++];
        }

        // Prepare Response

        //| SLAVE_ID | FUNCTION_CODE | Start Addr | num of Regs    | CRC     |
        //| 1 BYTE   |  1 BYTE       |  2 BYTE    | 2 BYTES      | 2 BYTES |

        TxData[0] = SLAVE_ID;    // slave ID
        TxData[1] = RxData[1];   // function code
        TxData[2] = RxData[2];   // Start Addr HIGH Byte
        TxData[3] = RxData[3];   // Start Addr LOW Byte
        TxData[4] = RxData[4];   // num of Regs HIGH Byte
        TxData[5] = RxData[5];   // num of Regs LOW Byte

        sendData(TxData, 6);  // send data... CRC will be calculated in the function itself
        return 1;   // success
}

uint8_t writeSingleReg(void)
{
   uint16_t coilAddr = ((RxData[2] << 8) | RxData[3]);  // Coil Address
   uint16_t coilValue = ((RxData[4] << 8) | RxData[5]);  // Coil Value
```

```
// Ensure the coil address is within valid range



if(coilAddr == RELAY_REGISTER_ADDRESS ){
// Handle coil value to set or reset coil
if (coilValue == 0xFF00) {
    // Reset coil (turn relay off)
    Coils_Database[coilAddr / 8] &= ~(1 << (coilAddr % 8));
    turnRelayOff();
 } else if (coilValue == 0x0000) {
    // Set coil (turn relay on)
    Coils_Database[coilAddr / 8] |= (1 << (coilAddr % 8));
    turnRelayOn();
 } else {
    modbusException(ILLEGAL_DATA_VALUE);  // Invalid value for a coil
    return 0;
 }


}


// Prepare response
TxData[0] = SLAVE_ID;  // Slave ID
TxData[1] = RxData[1];  // Function code
TxData[2] = RxData[2];  // Coil address high byte
```

```
    TxData[3] = RxData[3];  // Coil address low byte
    TxData[4] = RxData[4];  // Coil value high byte
    TxData[5] = RxData[5];  // Coil value low byte

    // Send response with CRC calculation
    sendData(TxData, 6);
    return 1;  // Success
}
```

ModbusSlave.h

```
#ifndef INC_MODBUSSLAVE_H_
#define INC_MODBUSSLAVE_H_

#include "modbus_crc.h"
#include "stm32f4xx_hal.h"

#define SLAVE_ID 0x07

#define ILLEGAL_FUNCTION       0x01
#define ILLEGAL_DATA_ADDRESS   0x02
#define ILLEGAL_DATA_VALUE     0x03
```

```
static uint16_t Holding_Registers_Database[50]={

        0000, 1111, 2222, 3333, 4444, 5555, 6666, 7777, 8888, 9999,  // 0-9 40001-40010

        12345, 15432, 15535, 10234, 19876, 13579, 10293, 19827, 13456, 14567,  // 10-19 40011-40020

        21345, 22345, 24567, 25678, 26789, 24680, 20394, 29384, 26937, 27654,  // 20-29 40021-40030

        31245, 31456, 34567, 35678, 36789, 37890, 30948, 34958, 35867, 36092,  // 30-39 40031-40040

        45678, 46789, 47890, 41235, 42356, 43567, 40596, 49586, 48765, 41029,  // 40-49 40041-40050

};


static const uint16_t Input_Registers_Database[50]={

        0000, 1111, 2222, 3333, 4444, 5555, 6666, 7777, 8888, 9999,  // 0-9 30001-30010

        12345, 15432, 15535, 10234, 19876, 13579, 10293, 19827, 13456, 14567,  // 10-19 30011-30020

        21345, 22345, 24567, 25678, 26789, 24680, 20394, 29384, 26937, 27654,  // 20-29 30021-30030

        31245, 31456, 34567, 35678, 36789, 37890, 30948, 34958, 35867, 36092,  // 30-39 30031-30040

        45678, 46789, 47890, 41235, 42356, 43567, 40596, 49586, 48765, 41029,  // 40-49 30041-30050

};


static uint8_t Coils_Database[25]={

        0b01001001, 0b10011100, 0b10101010, 0b01010101, 0b11001100,     // 0-39 1-40
```

```
                0b10100011, 0b01100110, 0b10101111, 0b01100000, 0b10111100,    // 40-79
41-80

                0b11001100, 0b01101100, 0b01010011, 0b11111111, 0b00000000,    // 80-
119  81-120

                0b01010101, 0b00111100, 0b00001111, 0b11110000, 0b10001111,    // 120-
159 121-160

                0b01010100, 0b10011001, 0b11111000, 0b00001101, 0b00101010,    // 160-
199 161-200

};



static const uint8_t Inputs_Database[25]={

                0b01001001, 0b10011100, 0b10101010, 0b01010101, 0b11001100,    // 0-39
10001-10040

                0b10100011, 0b01100110, 0b10101111, 0b01100000, 0b10111100,    // 40-79
10041-10080

                0b11001100, 0b01101100, 0b01010011, 0b11111111, 0b00000000,    // 80-
119  10081-10120

                0b01010101, 0b00111100, 0b00001111, 0b11110000, 0b10001111,    // 120-
159 10121-10160

                0b01010100, 0b10011001, 0b11111000, 0b00001101, 0b00101010,    // 160-
199 10161-10200

};




uint8_t readHoldingRegs (void);

uint8_t readInputRegs (void);



uint8_t readCoils (void);

uint8_t readInputs (void);
```

```
uint8_t writeSingleReg (void);
uint8_t writeHoldingRegs (void);
void turnRelayOn(void);
void turnRelayOff(void);

void modbusException (uint8_t exceptioncode);

#endif /* INC_MODBUSSLAVE_H_ */
```

Modbus_crc.c

```
#include "modbusSlave.h"
#include "string.h"
#include "main.h"
extern uint8_t RxData[256];
extern uint8_t TxData[256];
extern UART_HandleTypeDef huart2;

#define RELAY_REGISTER_ADDRESS 0x05


void turnRelayOn(void)
{
```

```
   HAL_GPIO_WritePin(Relay_out_GPIO_Port,  Relay_out_Pin,  GPIO_PIN_SET);      // Replace GPIOx and GPIO_PIN_y with actual GPIO port and pin

}


// Function to turn the relay off

void turnRelayOff(void)

{

   HAL_GPIO_WritePin(Relay_out_GPIO_Port,  Relay_out_Pin,  GPIO_PIN_RESET);    // Replace GPIOx and GPIO_PIN_y with actual GPIO port and pin

}


void sendData (uint8_t *data, int size)

{

        // we will calculate the CRC in this function itself

        uint16_t crc = crc16(data, size);

        data[size] = crc&0xFF;   // CRC LOW

        data[size+1] = (crc>>8)&0xFF;  // CRC HIGH


        HAL_UART_Transmit(&huart2, data, size+2, 1000);

}


void modbusException (uint8_t exceptioncode)

{

        //| SLAVE_ID | FUNCTION_CODE | Exception code | CRC     |

        //| 1 BYTE   |  1 BYTE       |    1 BYTE      | 2 BYTES |
```

```
	TxData[0] = RxData[0];		// slave ID
	TxData[1] = RxData[1]|0x80;  // adding 1 to the MSB of the function code
	TxData[2] = exceptioncode;   // Load the Exception code
	sendData(TxData, 3);		// send Data... CRC will be calculated in the function
}



uint8_t readHoldingRegs (void)
{
	uint16_t startAddr = ((RxData[2]<<8)|RxData[3]);  // start Register Address


	uint16_t numRegs = ((RxData[4]<<8)|RxData[5]);    // number to registers master has requested
	if ((numRegs<1)||(numRegs>125))  // maximum no. of Registers as per the PDF
	{
		modbusException (ILLEGAL_DATA_VALUE);  // send an exception
		return 0;
	}


	uint16_t endAddr = startAddr+numRegs-1;  // end Register
	if (endAddr>49)  // end Register can not be more than 49 as we only have record of 50 Registers in total
	{
		modbusException(ILLEGAL_DATA_ADDRESS);   // send an exception
		return 0;
	}
```

```
        // Prepare TxData buffer

        //| SLAVE_ID | FUNCTION_CODE | BYTE COUNT | DATA      | CRC     |
        //| 1 BYTE   |  1 BYTE       |  1 BYTE    | N*2 BYTES | 2 BYTES |

        TxData[0] = SLAVE_ID;  // slave ID
        TxData[1] = RxData[1];  // function code
        TxData[2] = numRegs*2;  // Byte count
        int indx = 3;  // we need to keep track of how many bytes has been stored in TxData Buffer

        for (int i=0; i<numRegs; i++)   // Load the actual data into TxData buffer
        {
                TxData[indx++]  =  (Holding_Registers_Database[startAddr]>>8)&0xFF;    // extract the higher byte
                TxData[indx++]  =  (Holding_Registers_Database[startAddr])&0xFF;       // extract the lower byte
                startAddr++;  // increment the register address
        }

        sendData(TxData, indx);  // send data... CRC will be calculated in the function itself
        return 1;   // success
}
```

```
uint8_t writeHoldingRegs (void)
{
	uint16_t startAddr = ((RxData[2]<<8)|RxData[3]);  // start Register Address

	uint16_t numRegs = ((RxData[4]<<8)|RxData[5]);    // number to registers master has requested
	if ((numRegs<1)||(numRegs>123))  // maximum no. of Registers as per the PDF
	{
		modbusException (ILLEGAL_DATA_VALUE);  // send an exception
		return 0;
	}

	uint16_t endAddr = startAddr+numRegs-1;  // end Register
	if (endAddr>49)  // end Register can not be more than 49 as we only have record of 50 Registers in total
	{
		modbusException(ILLEGAL_DATA_ADDRESS);   // send an exception
		return 0;
	}

	/* start saving 16 bit data
	 * Data starts from RxData[7] and we need to combine 2 bytes together
	 * 16 bit Data = firstByte<<8|secondByte
	 */
	int indx = 7;  // we need to keep track of index in RxData
	for (int i=0; i<numRegs; i++)
```

```
        {
                Holding_Registers_Database[startAddr++] = (RxData[indx++]<<8)|RxData[indx++];
        }

        // Prepare Response

        //| SLAVE_ID | FUNCTION_CODE | Start Addr | num of Regs    | CRC     |
        //| 1 BYTE   |  1 BYTE       |  2 BYTE    | 2 BYTES      | 2 BYTES |

        TxData[0] = SLAVE_ID;    // slave ID
        TxData[1] = RxData[1];   // function code
        TxData[2] = RxData[2];   // Start Addr HIGH Byte
        TxData[3] = RxData[3];   // Start Addr LOW Byte
        TxData[4] = RxData[4];   // num of Regs HIGH Byte
        TxData[5] = RxData[5];   // num of Regs LOW Byte

        sendData(TxData, 6);  // send data... CRC will be calculated in the function itself
        return 1;   // success
}

uint8_t writeSingleReg(void)
{
   uint16_t coilAddr = ((RxData[2] << 8) | RxData[3]);  // Coil Address
   uint16_t coilValue = ((RxData[4] << 8) | RxData[5]);  // Coil Value
```

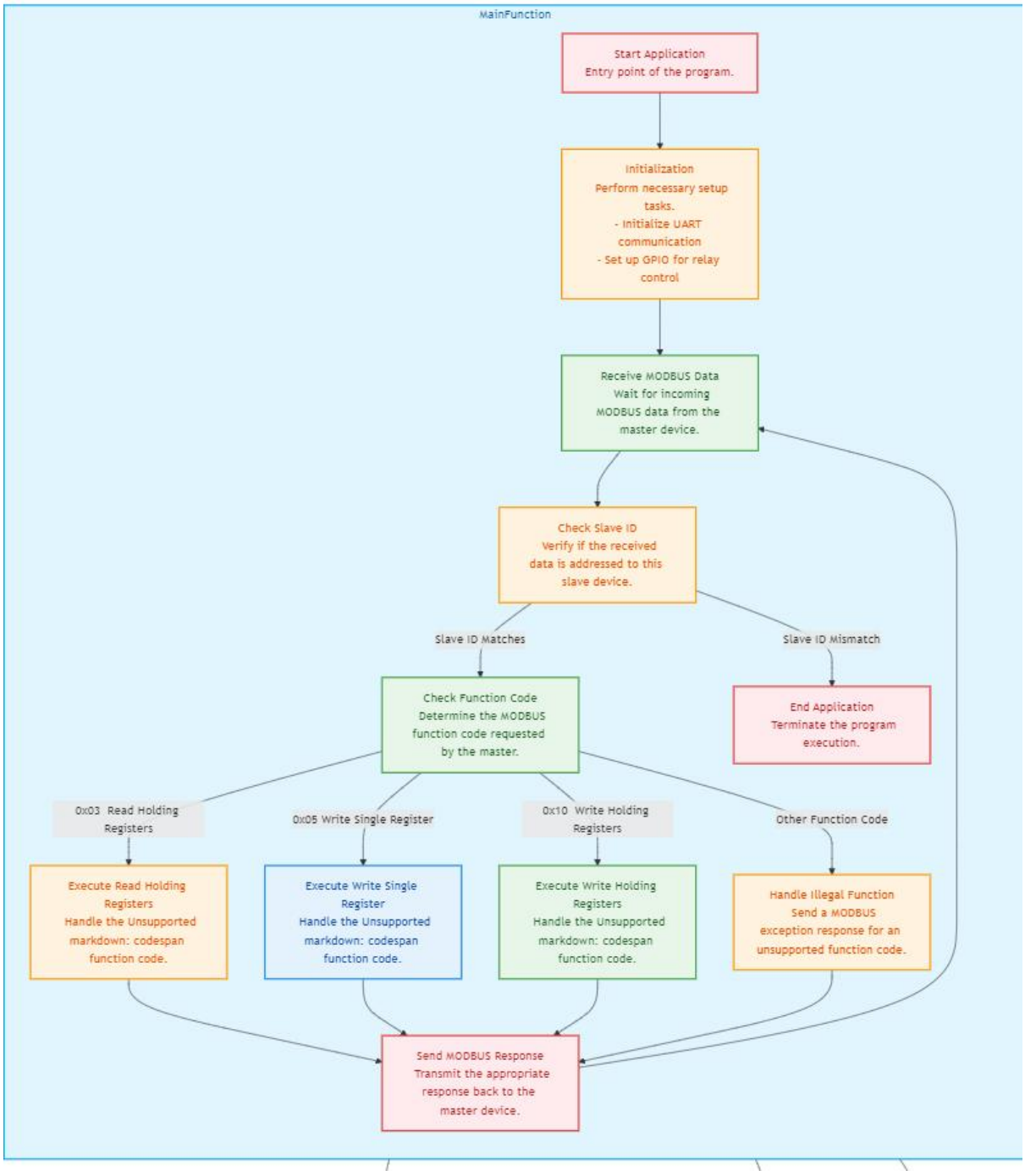
```
// Ensure the coil address is within valid range



if(coilAddr == RELAY_REGISTER_ADDRESS ){
// Handle coil value to set or reset coil
if (coilValue == 0xFF00) {
    // Reset coil (turn relay off)
    Coils_Database[coilAddr / 8] &= ~(1 << (coilAddr % 8));
    turnRelayOff();
 } else if (coilValue == 0x0000) {
    // Set coil (turn relay on)
    Coils_Database[coilAddr / 8] |= (1 << (coilAddr % 8));
    turnRelayOn();
 } else {
    modbusException(ILLEGAL_DATA_VALUE);  // Invalid value for a coil
    return 0;
 }


}


// Prepare response
TxData[0] = SLAVE_ID;  // Slave ID
TxData[1] = RxData[1];  // Function code
TxData[2] = RxData[2];  // Coil address high byte
```

```
    TxData[3] = RxData[3];  // Coil address low byte
    TxData[4] = RxData[4];  // Coil value high byte
    TxData[5] = RxData[5];  // Coil value low byte

    // Send response with CRC calculation
    sendData(TxData, 6);
    return 1;  // Success
}
```

Modbus_crc.h

```
#ifndef INC_MODBUS_CRC_H_
#define INC_MODBUS_CRC_H_
#include "stdint.h"
uint16_t crc16(uint8_t *buffer, uint16_t buffer_length);
#endif /* INC_MODBUS_CRC_H_ */
```

CODE FLOW


MainFunction
Start Application
Entry point of the program.
Initialization
Perform necessary setup tasks.
- Initialize UART communication
- Set up GPIO for relay control
Receive MODBUS Data
Wait for incoming MODBUS data from the master device.
Check Slave ID
Verify if the received data is addressed to this slave device.
Slave ID Matches
Slave ID Mismatch
End Application
Terminate the program execution.
Check Function Code
Determine the MODBUS function code requested by the master.
0x03 Read Holding Registers
0x05 Write Single Register
0x10 Write Holding Registers
Other Function Code
Execute Read Holding Registers
Handle the Unsupported markdown: codespan function code.
Execute Write Single Register
Handle the Unsupported markdown: codespan function code.
Execute Write Holding Registers
Handle the Unsupported markdown: codespan function code.
Handle Illegal Function
Send a MODBUS exception response for an unsupported function code.
Send MODBUS Response
Transmit the appropriate response back to the master device.

FUTURE ENCHANCEMENT

Integration with Additional Sensors:

- Expand the system to include more environmental sensors (e.g., humidity, light, CO2) to monitor a wider range of conditions.
- Implement multi-sensor fusion to make more informed decisions based on combined data from various sensors.

Advanced Data Analytics and AI:

- Incorporate machine learning algorithms to predict temperature trends and automatically adjust the critical temperature threshold based on historical data.
- Use AI to optimize energy usage by controlling the external device more efficiently based on environmental conditions and patterns.

Expansion to Multi-Device Networks:

- Scale the system to manage multiple RuggedBoards and STM32 microcontrollers in larger, distributed networks for industrial or agricultural applications.
- Implement a centralized dashboard to monitor and control multiple systems across different locations from a single PhyCloud interface.

Energy Efficiency Optimization:

- Integrate energy-efficient algorithms to optimize the operation of the external device, reducing power consumption based on real-time data.
- Incorporate renewable energy sources (e.g., solar panels) to power the system, making it more sustainable and self-sufficient.

Cloud Data Storage and Historical Analysis:

- Store temperature and device status data on the cloud for long-term analysis and reporting.
- Provide tools for generating reports and visualizing historical data trends to improve decision-making processes.

Support for Multiple Communication Protocols:

- Add support for other industrial communication protocols (e.g., CAN, BACnet) to increase compatibility with a wider range of devices.
- Implement wireless communication options (e.g., Wi-Fi, LoRa) for environments where wired connections are not feasible.

CONCLUSION

This project successfully demonstrates the integration of IoT technology with industrial automation by using a RuggedBoard A5D2X as the master device and STM32 F446RE microcontrollers as slaves. The system efficiently monitors temperature using a KY-013 sensor and controls an external device via a relay based on a critical temperature threshold. Communication between the master and slaves is handled robustly through the Modbus protocol over an RS485 network, ensuring reliable data transmission. The connection to PhyCloud MQTT further enhances the system by providing remote monitoring and control capabilities, making the setup both flexible and scalable for future enhancements.