**SMART GREENHOUSE**

**BY GROUP 2**

**AKHIL R**

**BOOBATHI K**

**TABLE OF CONTENTS**

1 PROJECT OVERVIEW

2 HARDWARE SPECIFICATIONS

3 BLOCK DIAGRAM

4 CODE FLOW

5 SOURCE CODE

6 RESULT

7 CONCLUSION

**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.

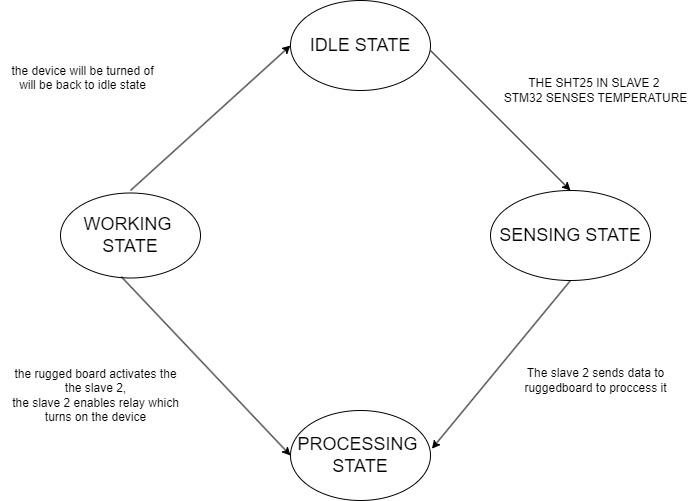
**8. Safety and Compliance:**

* **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**

****

.#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();

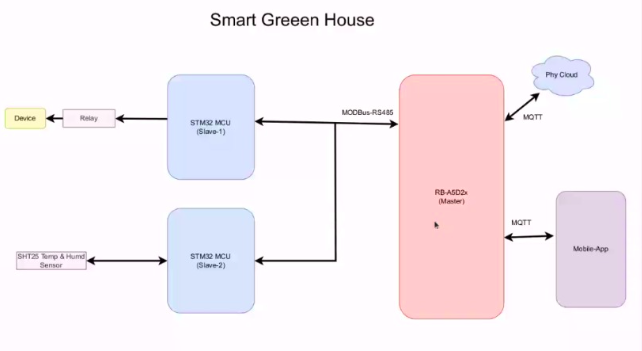
initMQTT();

mainLoop();

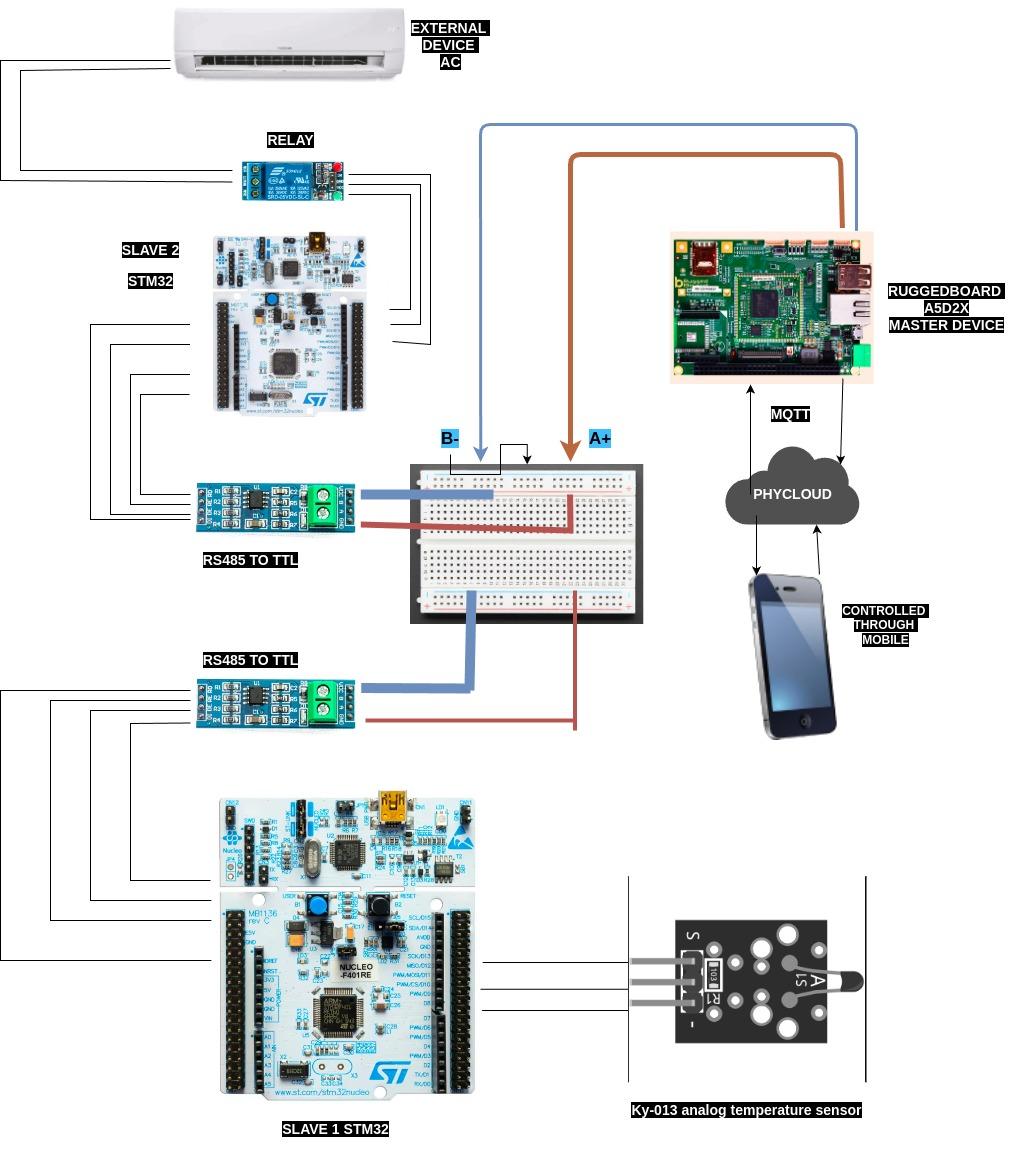
return 0;

}

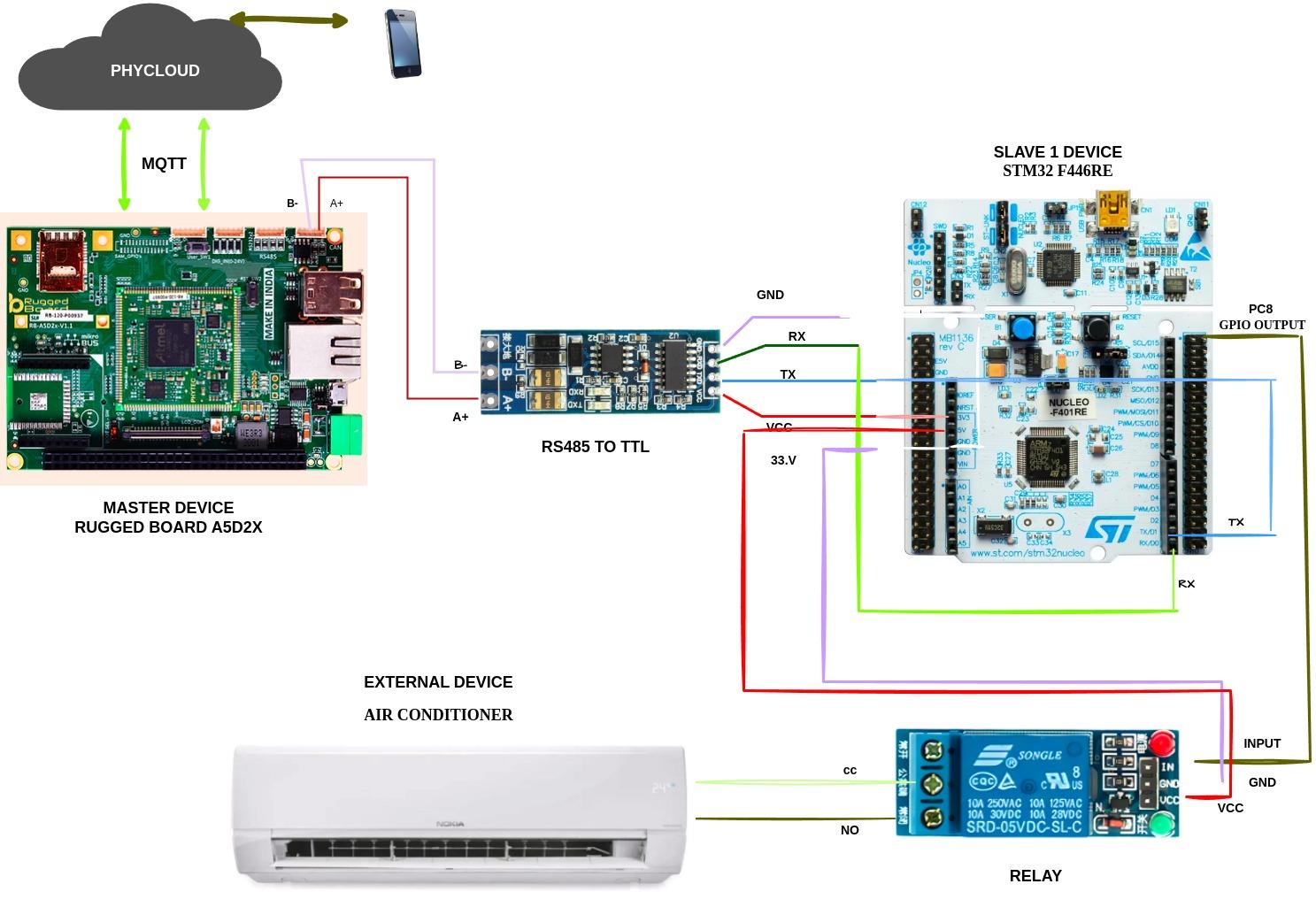
**HIGH LEVEL BLOCK DIAGRAM**

****

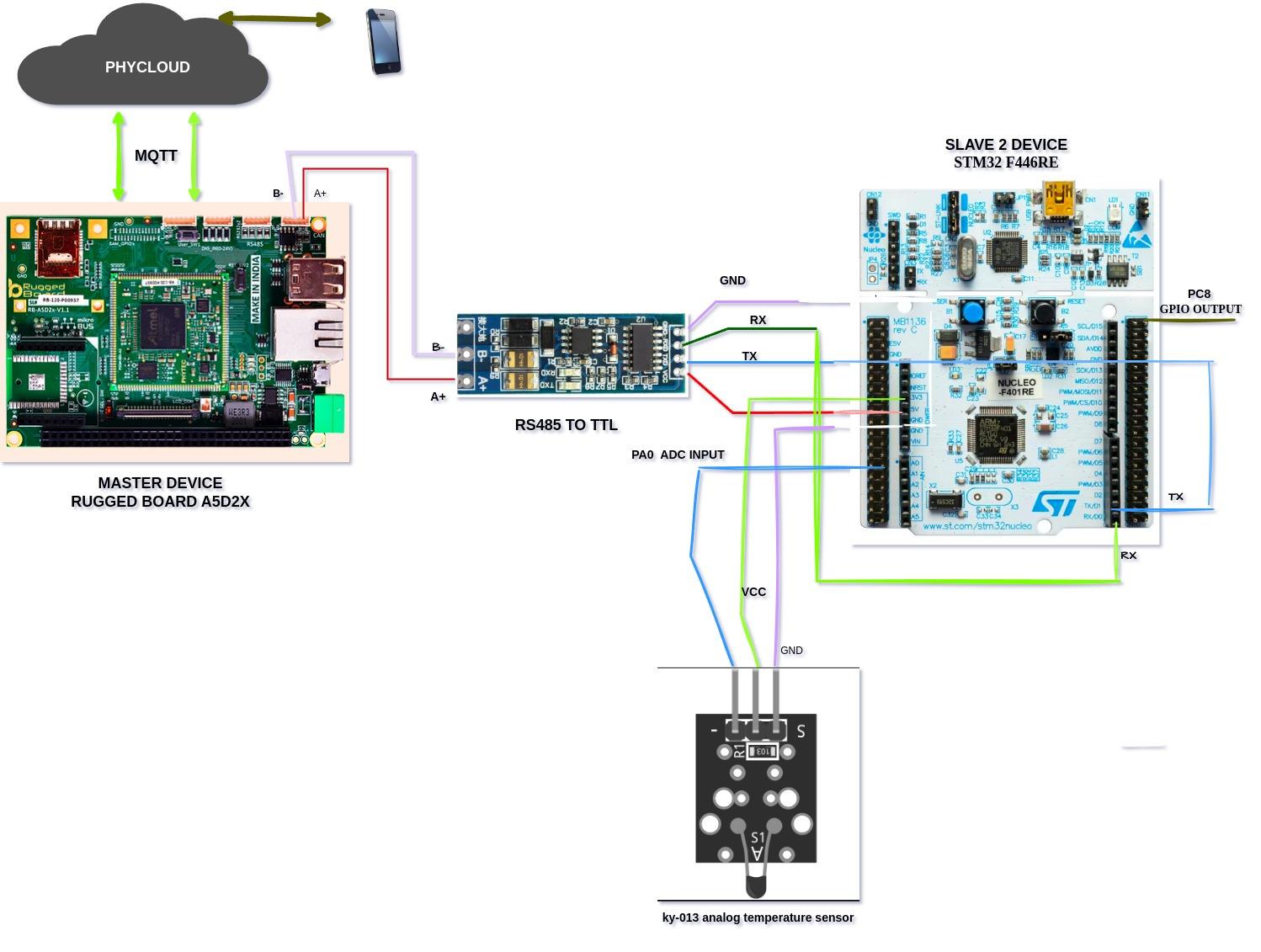
**LOW LEVEL BLOCK DIAGRAM**



**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.**

**CODE**

**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 Device 2 stm32 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\_SCALE1);**

**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\_**

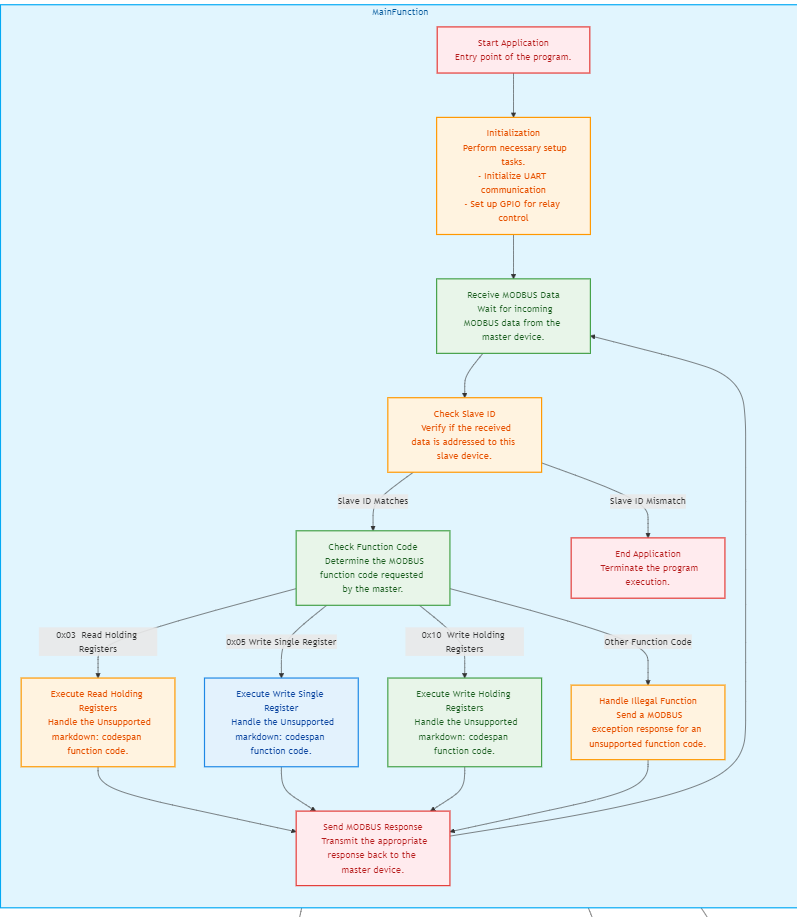
**#define INC\_MODBUS\_CRC\_H\_**

**#include "stdint.h"**

**uint16\_t crc16(uint8\_t \*buffer, uint16\_t buffer\_length);**

**#endif /\* INC\_MODBUS\_CRC\_H\_ \*/**

**CODE FLOW**

****

**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.**