INDUSTRY ENERGY MONITERING SYSTEM

DEVA N

SURYA M

SANJEEVI RAM N

Industrial Energy Monitoring System

📰 Project Summary

This project focuses on developing an advanced industrial energy monitoring system that efficiently captures, processes, and stores energy usage data for real-time analysis and long-term retention. The system starts by reading analog signals from industrial loads, which are converted into digital data using an external Modbus RTU device with 8 input channels. The STM32F446RE microcontroller acts as the central processing unit, interfacing with the Modbus RTU device to retrieve and manage the digital data, which is then processed for transmission.

Wireless communication is enabled through a BE33 Bluetooth module connected to the STM32 microcontroller. This module transmits the processed data to a Rugged Board (RB-A5D2x) with Bluetooth capability. Once the data is received by the Rugged Board, it is sent to cloud storage for remote access and real-time analysis. The cloud storage also allows long-term data retention, enabling the monitoring of energy consumption patterns over time.

The system is designed to optimize industrial energy usage by providing detailed insights into energy consumption, allowing operators to make informed decisions that can improve efficiency and reduce operational costs. By integrating technologies such as Modbus RTU, STM32 microcontroller, Bluetooth communication, and cloud storage, this project delivers a comprehensive solution for industrial energy monitoring suited to modern, data-driven industrial environments.

The project aims to develop an advanced industrial energy monitoring system that captures, processes, and stores energy usage data for real-time analysis and long-term retention. It uses a Modbus RTU device to convert analog signals to digital data, managed by an STM32F446RE microcontroller. Data is transmitted wirelessly via a BE33 Bluetooth module to a Rugged Board, which sends it to cloud storage for remote access and analysis. The system seeks to optimize energy usage, improve efficiency, and reduce operational costs by providing detailed insights into energy consumption.

💡 Objective

In order to maximize energy efficiency and lower operating costs, an advanced industrial energy monitoring system that gathers, processes, and retains real-time energy usage data for analysis and long-term retention is being developed. The system will use Bluetooth for wireless data transfer, STM32 microcontroller processing, Modbus RTU connection, and cloud storage integration to deliver comprehensive and useful insights into energy use trends.

Block Diagram

Connection Diagram

Low Level Block Diagram

Code Flow

Start
Initialize UART and FSM
Check_BE33_connection()
STATE_IDLE
Send CMD?NAME and wait for response
FALSE
Validate response and transition to next state
TRUE
STATE_SEND_CMD_RESET
Send CMD_RESET=0 and wait for response
Similar flow for other states
FALSE
Validate response and transition to next state
TRUE
STATE_SEND_CMD_DATA
Send CMD_DATA=<conn_handle>, data
EnergyMeterValue()
Read Modbus parameters and send data

State Block Diagram

Finite State Machine

Implementing code using a state machine is an extremely handy design technique for solving complex engineering problems. State machines break down the design into a series of steps, or what are called states in state-machine lingo. Each state performs some narrowly defined task. Events, on the other hand, are the stimuli, which cause the state machine to move, or transition, between states.

1. **Idle** — the motor is not spinning but is at rest
    - Do nothing
2. **Start** — starts the motor from a dead stop
    - Turn on motor power
    - Set motor speed
3. **Change Speed** — adjust the speed of an already moving motor
    - Change motor speed
4. **Stop** — stop a moving motor
    - Turn off motor power
    - Go to the Idle state

For Example

```
typedef enum {
    STATE_IDLE,
    STATE_SEND_CMD_NAME,
    STATE_WAIT_CMD_NAME,
    STATE_SEND_CMD_RESET,
    STATE_WAIT_CMD_RESET,
    STATE_SEND_CMD_TXPWR,
    STATE_WAIT_CMD_TXPWR,
    STATE_SEND_CMD_SCANPARAM,
    STATE_WAIT_CMD_SCANPARAM,
    STATE_SEND_CMD_SCAN,
    STATE_WAIT_CMD_SCAN,
    STATE_SEND_CMD_CON,
    STATE_WAIT_CMD_CON,
    STATE_SEND_CMD_DATA,
    STATE_WAIT_CMD_DATA,
        STATE_SEND_CMD_VALUE,
        STATE_WAIT_CMD_VALUE
} FSM_State;

//use this state in switch case
switch (currentState) {
   case ST_IDLE:
       // do something in the idle state
       break;

    case ST_STOP:
       // do something in the stop state
       break;

    // etc...
}
```

Industrial Energy Monitor System With STM32

STM32F446RE

The STM32 Nucleo board is an affordable and flexible development platform that allows users to prototype with various STM32 microcontrollers. It features Arduino™ connectivity support and ST Morpho headers, which enable easy expansion with specialized shields. The board comes with an integrated ST-LINK/V2-1 debugger/programmer, eliminating the need for a separate probe.

Microcontroller Highlights:

- **Core**: Arm 32-bit Cortex-M4 CPU with FPU, running at up to 180 MHz, with adaptive real-time acceleration.
- **Memory**: Includes 512 KB of Flash memory, 128 KB of SRAM, and support for external memory interfaces.
- **Clock & Power**: Supports a wide range of supply voltages, internal and external oscillators, and low-power modes like Sleep, Stop, and Standby.
- **Peripherals**:
  - **ADC/DAC**: Three 12-bit ADCs and two 12-bit DACs.
  - **Timers**: Up to 17 timers, including watchdogs, SysTick, and multiple 16- and 32-bit timers.
  - **I/O Ports**: Up to 114 I/O ports, with high-speed and 5V-tolerant options.
  - **Communication**: Multiple interfaces including SPI, I2C, USART/UART, CAN, USB, and SDIO.
  - **Advanced Features**: USB 2.0 support, parallel camera interface, and real-time clock (RTC) with hardware calendar.

Energy Monitor System(RISH LM-1360)

The RISH LM-1360 is a sophisticated electrical parameter measuring instrument designed for monitoring and analyzing power quality in various electrical systems. It can measure a wide range of electrical parameters including voltage, current, frequency, power, energy, and power factor.

Key Features:

- **Multifunctional**: Capable of measuring and displaying multiple electrical parameters in real-time.
- **High Accuracy**: Provides precise measurements, making it suitable for critical applications.
- **User-Friendly Interface**: Equipped with an easy-to-read display and intuitive controls.
- **Data Logging**: Supports data logging, allowing users to store and analyze historical data.
- **Communication Capabilities**: Often features communication ports like RS485 for integration into larger monitoring systems.

- **Compact Design**: Designed to fit easily into control panels and switchboards.

TTL to RS485

A TTL to RS485 converter is a device that enables communication between devices that use Transistor-Transistor Logic (TTL) and those that use the RS485 standard.

Key Features:

- **Signal Conversion**: Converts TTL level signals (typically 0V and 5V or 3.3V) to RS485 differential signals, which are used for long-distance and noise-resistant communication.
- **Long-Distance Communication**: RS485 supports communication over long distances (up to 1200 meters) and can handle up to 32 devices on a single bus, making it ideal for industrial environments.
- **Differential Signaling**: RS485 uses differential signaling to reduce noise and interference, which is crucial for reliable communication in electrically noisy environments.
- **Multi-Device Support**: RS485 allows multiple devices to communicate on the same bus, enabling complex network configurations.
- **Common Applications**: Used in industrial automation, data acquisition systems, and other applications where reliable, long-distance communication is needed.

RuggedBoard BLU-BE33

The **BE33 Bluetooth module** is a versatile, low-power Bluetooth device designed for wireless communication in various applications. Here's a summary of its features and capabilities:

Key Features:

- **Bluetooth Version**: The BE33 typically supports Bluetooth 4.0 or later, enabling efficient wireless communication with a wide range of devices.
- **Communication Range**: It offers a decent range, usually up to 10 meters (33 feet), suitable for short-range wireless connections.
- **UART Interface**: The module uses UART (Universal Asynchronous Receiver/Transmitter) for communication with microcontrollers, making it easy to integrate into embedded systems.
- **Low Power Consumption**: Optimized for low energy consumption, making it ideal for battery-powered devices.
- **AT Commands**: The module can be configured using AT commands, allowing for control over various settings such as baud rate, pairing, and mode (master/slave).
- **Compact Size**: Small and easy to integrate into projects where space is a concern.
- **Applications**: Commonly used in wireless data transmission, Bluetooth serial communication, IoT projects, and remote control systems.

Applications:

- **Wireless Data Transmission**: Ideal for sending and receiving data wirelessly between devices.
- **Bluetooth Serial Communication**: Facilitates communication between microcontrollers and other Bluetooth-enabled devices.
- **IoT Projects**: Used in Internet of Things (IoT) devices for wireless communication.
- **Remote Control Systems**: Can be used in remote-controlled devices and systems.

Project code module

1. Overview

This code implements an energy meter system using an STM32 microcontroller. It communicates with a Modbus device to read energy parameters and transmits this data via Bluetooth using a BE33 module. The system uses a finite state machine (FSM) to manage the connection and data transmission process.

2. Key Components

2.1 Hardware

- STM32 Microcontroller
- Modbus-compatible Energy Meter
- BE33 Bluetooth Module

2.2 Software

- STM32 HAL Library
- Custom Modbus CRC calculation
- IEEE 754 float conversion

3. Main Functions

3.1 main()

Initializes the system, peripherals, and starts the main loop.

3.2 check_BE33_connection()

Implements the FSM for Bluetooth connection and data transmission. It handles the following states:

- Sending and verifying device name
- Resetting the device
- Setting transmission power
- Configuring scan parameters

- Scanning for devices
- Connecting to a specific device
- Sending data

3.3 EnergyMeterValue()

Reads various energy parameters from the Modbus device and sends them via Bluetooth. Parameters include:

- Voltage
- Current
- Frequency
- Power
- Power Factor
- Energy
- RPM

3.4 ReadModbusParameter()

Sends a Modbus request to read a specific parameter and processes the response.

3.5 ieee754_to_float()

Converts IEEE 754 formatted data to a float value.

4. Utility Functions

- sendCommand(): Sends commands to the BE33 module
- processResponse(): Processes responses from the BE33 module
- validateResponse(): Checks if the received response matches the expected one

5. Interrupt Handlers

HAL_UART_RxCpltCallback(): Handles UART receive interrupts, storing received data in a buffer.

6. Error Handling

The code implements a retry mechanism for failed commands, with a maximum retry count of 3.

7. Debugging

The code uses ITM_SendChar() for debugging output, which can be viewed using appropriate debugging tools.

8. Notes

- The code uses blocking delays (HAL_Delay()) which may affect real-time performance.
- Modbus communication is done over UART3, while Bluetooth communication uses UART1.
- The Bluetooth device address is hardcoded (fab321a20744).

💡 Enable USART1 for BE33 → choose Asynchronous and enable hardware flow control for both→ click interrupt
💡 Enable USART3 for Energy Monitor System(Rish lm-1340) →choose Asynchronous → set bardrate 9600

💡 Enable the setting to print float value

Output View:

💡 Enable SWV Date console

💡 Select port 0 to display the output

💡 To check the transmitted output on cutecom using USB to TTL

CODE:

```
//main.c

/* Includes ------------------------------------------------------------------*/

#include "main.h"

#include "string.h"

#include "stdio.h"

#include "modbus_crc.h"


/* Private typedef -----------------------------------------------------------*/

/* USER CODE BEGIN PTD */


float ieee754_to_float(uint8_t byte1, uint8_t byte2, uint8_t byte3,uint8_t byte4);

float ReadModbusParameter(uint8_t regAddressHigh, uint8_t regAddressLow, uint8_t
regCountHigh, uint8_t regCountLow);


/* USER CODE END PTD */


/* Private macro -------------------------------------------------------------*/

/* USER CODE BEGIN PM */


uint8_t rxData[9];

uint8_t txData[8]; // Function code 0x03 for holding registers


/* USER CODE END PM */
```

```
/* Private variables ---------------------------------------------------------*/

UART_HandleTypeDef huart1;
UART_HandleTypeDef huart2;
UART_HandleTypeDef huart3;

/* USER CODE BEGIN PV */

typedef enum {
    STATE_IDLE,
    STATE_SEND_CMD_NAME,
    STATE_WAIT_CMD_NAME,
    STATE_SEND_CMD_RESET,
    STATE_WAIT_CMD_RESET,
    STATE_SEND_CMD_TXPWR,
    STATE_WAIT_CMD_TXPWR,
    STATE_SEND_CMD_SCANPARAM,
    STATE_WAIT_CMD_SCANPARAM,
    STATE_SEND_CMD_SCAN,
    STATE_WAIT_CMD_SCAN,
    STATE_SEND_CMD_CON,
    STATE_WAIT_CMD_CON,
    STATE_SEND_CMD_DATA,
    STATE_WAIT_CMD_DATA,
        STATE_SEND_CMD_VALUE,
```

```
            STATE_WAIT_CMD_VALUE
} FSM_State;

FSM_State currentState = STATE_IDLE;

#define RX_BUFFER_SIZE 300

uint8_t rxBuffer[RX_BUFFER_SIZE]; // Buffer for receiving data
uint8_t responseBuffer[RX_BUFFER_SIZE];
volatile uint16_t bufferIndex = 0;
volatile uint8_t rxComplete = 0;

int commandRetryCount = 0; // Counter for retry attempts
#define MAX_RETRY_COUNT 3    // Maximum number of retry attempts

/* USER CODE END PV */

/* Private function prototypes -----------------------------------------------*/
void SystemClock_Config(void);
static void MX_GPIO_Init(void);
static void MX_USART2_UART_Init(void);
static void MX_USART1_UART_Init(void);
static void MX_USART3_UART_Init(void);
/* USER CODE BEGIN PFP */
void sendCommand(const char *cmd);
```

```
void processResponse(void);
int validateResponse(const char *expectedResponse);

/* USER CODE END PFP */

int main(void)
{

  HAL_Init();



  SystemClock_Config();



  /* Initialize all configured peripherals */
  MX_GPIO_Init();
  MX_USART2_UART_Init();
  MX_USART1_UART_Init();
  MX_USART3_UART_Init();

  // Start UART receive interrupt
  HAL_UART_Receive_IT(&huart1, rxBuffer, 1);

  currentState = STATE_SEND_CMD_NAME; // Start FSM by sending the first command
```

```
  while (1)
  {

                    if (rxComplete) {
                              processResponse();
                    }

                    check_BE33_connection();

  }

}

void check_BE33_connection(void) {
    switch (currentState) {
        case STATE_SEND_CMD_NAME:
            sendCommand("CMD?NAME\r\n");
            HAL_Delay(1000); // Replace with non-blocking timer in production
            currentState = STATE_WAIT_CMD_NAME;
            break;

        case STATE_WAIT_CMD_NAME:
            if (validateResponse("RSP=0")) {
                currentState = STATE_SEND_CMD_RESET;
                commandRetryCount = 0;
```

```
    } else {
        if (commandRetryCount < MAX_RETRY_COUNT) {
            commandRetryCount++;
            sendCommand("CMD?NAME\r\n");
            HAL_Delay(1000);
        } else {
            // Handle retry failure (e.g., log error, alert user)
            currentState = STATE_IDLE;
        }
    }
    break;

case STATE_SEND_CMD_RESET:
    sendCommand("CMD+LREN=1\r\n");
    HAL_Delay(1000);
    sendCommand("CMD+RESET=0\r\n");
    HAL_Delay(1000);
    currentState = STATE_WAIT_CMD_RESET;
    break;

case STATE_WAIT_CMD_RESET:
    if (validateResponse("EVT+READY")) {
        currentState = STATE_SEND_CMD_TXPWR;
        commandRetryCount = 0;
    } else {
```

```
        if (commandRetryCount < MAX_RETRY_COUNT) {
            commandRetryCount++;
            sendCommand("CMD+RESET=0\r\n");
            HAL_Delay(1000);
        } else {
            // Handle retry failure
            currentState = STATE_IDLE;
        }
    }
    break;

case STATE_SEND_CMD_TXPWR:
    sendCommand("CMD+TXPWR=-4\r\n");
    HAL_Delay(1000);
    currentState = STATE_WAIT_CMD_TXPWR;
    break;

case STATE_WAIT_CMD_TXPWR:
    if (validateResponse("RSP=0")) {
        currentState = STATE_SEND_CMD_SCANPARAM;
        commandRetryCount = 0;
    } else {
        if (commandRetryCount < MAX_RETRY_COUNT) {
            commandRetryCount++;
            sendCommand("CMD+TXPWR=-4\r\n");
```

```
            HAL_Delay(1000);
        } else {
            // Handle retry failure
            currentState = STATE_IDLE;
        }
    }
    break;

case STATE_SEND_CMD_SCANPARAM:
    sendCommand("CMD+SCANPARAM=0,50,100,10000\r\n");
    HAL_Delay(1000);
    currentState = STATE_WAIT_CMD_SCANPARAM;
    break;

case STATE_WAIT_CMD_SCANPARAM:
    if (validateResponse("RSP=0")) {
        currentState = STATE_SEND_CMD_SCAN;
        commandRetryCount = 0;
    } else {
        if (commandRetryCount < MAX_RETRY_COUNT) {
            commandRetryCount++;
            sendCommand("CMD+SCANPARAM=0,50,100,10000\r\n");
            HAL_Delay(1000);
        } else {
            // Handle retry failure
```

```
            currentState = STATE_IDLE;
        }
    }
    break;

case STATE_SEND_CMD_SCAN:
    sendCommand("CMD+SCAN=1\r\n");
    HAL_Delay(6000);
    currentState = STATE_WAIT_CMD_SCAN;
    break;

case STATE_WAIT_CMD_SCAN:
    if (validateResponse("EVT+ADVRPT")) {
        currentState = STATE_SEND_CMD_CON;
        commandRetryCount = 0;
    } else {
        if (commandRetryCount < MAX_RETRY_COUNT) {
            commandRetryCount++;
            sendCommand("CMD+SCAN=1\r\n");
            HAL_Delay(6000);
        } else {
            // Handle retry failure
            currentState = STATE_IDLE;
        }
    }
```

```
    break;

case STATE_SEND_CMD_CON:
    sendCommand("CMD+CON=1,fab321a20744\r\n");
    HAL_Delay(10000);
    currentState = STATE_WAIT_CMD_CON;
    break;

case STATE_WAIT_CMD_CON:
    if (validateResponse("EVT+CON") && validateResponse("EVT+NOTIFY")) {
        currentState = STATE_SEND_CMD_DATA;
        commandRetryCount = 0;
    } else {
        if (commandRetryCount < MAX_RETRY_COUNT) {
            commandRetryCount++;
            sendCommand("CMD+CON=1,fab321a20744\r\n");
            HAL_Delay(10000);
        } else {
            // Handle retry failure
            currentState = STATE_IDLE;
        }
    }
    break;

case STATE_SEND_CMD_DATA:
```

```
            sendCommand("CMD+DATA=<conn_handle>,WELCOME TO EVERY ONE\r\n");
            HAL_Delay(1000);

            currentState = STATE_SEND_CMD_VALUE;
            break;

        case STATE_SEND_CMD_VALUE:
            EnergyMeterValue();
            currentState = STATE_SEND_CMD_VALUE;
            break;

        case STATE_IDLE:
        default:
            break;
    }
}



void sendCommand(const char *cmd) {
    HAL_UART_Transmit(&huart1, (uint8_t*) cmd, strlen(cmd), HAL_MAX_DELAY);
}

void processResponse(void) {
    // Print the entire response buffer to debug
```

```
for (uint16_t i = 0; i < bufferIndex; i++) {
    ITM_SendChar(responseBuffer[i]);
}
ITM_SendChar('\n'); // Newline for clarity

// Check and handle different responses based on state
switch (currentState) {
    case STATE_WAIT_CMD_NAME:
        if (validateResponse("RSP=0")) {
            currentState = STATE_SEND_CMD_RESET;
            commandRetryCount = 0;
        } else {
            // Handle unexpected response or retry
        }
        break;

    case STATE_WAIT_CMD_RESET:
        if (validateResponse("EVT+READY")) {
            currentState = STATE_SEND_CMD_TXPWR;
            commandRetryCount = 0;
        } else {
            // Handle unexpected response or retry
        }
        break;
```

```
case STATE_WAIT_CMD_TXPWR:
    if (validateResponse("RSP=0")) {
        currentState = STATE_SEND_CMD_SCANPARAM;
        commandRetryCount = 0;
    } else {
        // Handle unexpected response or retry
    }
    break;

case STATE_WAIT_CMD_SCANPARAM:
    if (validateResponse("RSP=0")) {
        currentState = STATE_SEND_CMD_SCAN;
        commandRetryCount = 0;
    } else {
        // Handle unexpected response or retry
    }
    break;

case STATE_WAIT_CMD_SCAN:
    if (validateResponse("EVT+ADVRPT")) {
        currentState = STATE_SEND_CMD_CON;
        commandRetryCount = 0;
    } else {
        // Handle unexpected response or retry
    }
```

```
            break;
        case STATE_WAIT_CMD_CON:
            if (validateResponse("EVT+CON")) {
                currentState = STATE_SEND_CMD_DATA;
                commandRetryCount = 0;
            } else {
                // Handle unexpected response or retry
            }
            break;
        case STATE_WAIT_CMD_DATA:
            // Handle CMD+DATA responses if necessary
            break;

        default:
            break;
    }
    // Clear buffer and reset index
    bufferIndex = 0;
    memset(responseBuffer, 0, RX_BUFFER_SIZE);
    rxComplete = 0;
}

void EnergyMeterValue(void){
    char cmd[100];
    float voltage = ReadModbusParameter(0x00, 0x00, 0x00, 0x02);
```

```
    snprintf(cmd, sizeof(cmd), "CMD+DATA=<conn_handle>,Voltage: %.2f V\r\n", voltage);

    sendCommand(cmd);

    HAL_Delay(1000);

    // Read and print current

    float current = ReadModbusParameter(0x00, 0x06, 0x00, 0x02);

    snprintf(cmd, sizeof(cmd), "CMD+DATA=<conn_handle>,Current: %.2f A\r\n", current);

    sendCommand(cmd);

    HAL_Delay(1000);

    // Read and print frequency

    float frequency = ReadModbusParameter(0x00, 0x46, 0x00, 0x02);

    snprintf(cmd, sizeof(cmd), "CMD+DATA=<conn_handle>,Frequency: %.2f Hz\r\n",
frequency);

    sendCommand(cmd);

    HAL_Delay(1000);

    // Read and print power

    float power = ReadModbusParameter(0x00, 0x0C, 0x00, 0x02);

     snprintf(cmd, sizeof(cmd), "CMD+DATA=<conn_handle>,power: %.2f Watt\r\n",
power);

    sendCommand(cmd);

    HAL_Delay(1000);

    // Read and print power factor

    float powerFactor = ReadModbusParameter(0x00, 0x1E, 0x00, 0x02);

    snprintf(cmd, sizeof(cmd), "CMD+DATA=<conn_handle>,powerFactor: %.2f \r\n",
powerFactor);

    sendCommand(cmd);

    HAL_Delay(1000);

    // Read and print Energy
```

```
    float energy = ReadModbusParameter(0x00, 0x6E, 0x00, 0x02);

    snprintf(cmd, sizeof(cmd), "CMD+DATA=<conn_handle>,Energy: %.2f KWh\r\n",
energy/1000);

    sendCommand(cmd);

    HAL_Delay(1000);

    // Read and print Energy

    float rpm = ReadModbusParameter(0x00, 0x88, 0x00, 0x02);

            snprintf(cmd, sizeof(cmd), "CMD+DATA=<conn_handle>,RMP: %.2f RPM\r\n",
rpm);

            sendCommand(cmd);

            HAL_Delay(1000);

}

int validateResponse(const char *expectedResponse) {

    // Check if the response buffer contains the expected response

    return strstr((char*) responseBuffer, expectedResponse) != NULL;

}



void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart) {

    if (huart->Instance == USART1) {

        if (bufferIndex < RX_BUFFER_SIZE - 1) {

            responseBuffer[bufferIndex++] = rxBuffer[0];

        }

        if (rxBuffer[0] == '\n' || bufferIndex >= RX_BUFFER_SIZE - 1) {

            responseBuffer[bufferIndex] = '\0';

            rxComplete = 1;

        }
```

```
        HAL_UART_Receive_IT(&huart1, rxBuffer, 1);

    }

}

// Function to send a Modbus request and receive the response

float ReadModbusParameter(uint8_t regAddressHigh, uint8_t regAddressLow, uint8_t
regCountHigh, uint8_t regCountLow) {

    uint8_t txData[8];

    uint8_t rxData[9];



    txData[0] = 0x01; // Slave Address

    txData[1] = 0x03; // Function Code

    txData[2] = regAddressHigh;

    txData[3] = regAddressLow;

    txData[4] = regCountHigh;

    txData[5] = regCountLow;



    uint16_t crc = crc16(txData, 6);

    txData[6] = crc & 0xFF;

    txData[7] = (crc >> 8) & 0xFF;



    if (HAL_UART_Transmit(&huart3, txData, sizeof(txData), 1000) == HAL_OK) {

        //Print_Raw_Data(txData, sizeof(txData));

        if (HAL_UART_Receive(&huart3, rxData, sizeof(rxData), 1000) == HAL_OK) {

            //Print_Raw_Data(rxData, sizeof(rxData));

            uint8_t byte1 = rxData[3];
```

```
            uint8_t byte2 = rxData[4];
            uint8_t byte3 = rxData[5];
            uint8_t byte4 = rxData[6];
            return ieee754_to_float(byte1, byte2, byte3, byte4);
        } else {
            printf("Failed to receive data\n");
            return -1;
        }
    } else {
        printf("Failed to transmit data\n");
        return -1;
    }
}

// Function to convert 32-bit IEEE 754 floating-point number to float
float ieee754_to_float(uint8_t byte1, uint8_t byte2, uint8_t byte3, uint8_t byte4) {
    uint32_t raw_value = (byte1 << 24) | (byte2 << 16) | (byte3 << 8) | byte4;
    int sign = (raw_value >> 31) & 0x1;
    int exponent = (raw_value >> 23) & 0xFF;
    uint32_t mantissa = raw_value & 0x7FFFFF;
    float normalized_mantissa = 1 + (float)mantissa / (1 << 23);
    float final_value = normalized_mantissa * (1 << (exponent - 127));
    if (sign) {
        final_value = -final_value;
    }
```

```
    return final_value;

}

int _write(int file, char *ptr, int len) {

    (void) file;

    int DataIdx;

    for (DataIdx = 0; DataIdx < len; DataIdx++) {

        ITM_SendChar(*ptr++);

    }

    return len;

}
```

CRC16 Calculation:

CRC16 (Cyclic Redundancy Check 16) is an error-detecting code used to detect accidental changes to raw data. This document describes the implementation of a CRC16 calculation using lookup tables for high-order and low-order bytes.

The purpose of this code is to calculate the CRC16 checksum for a given buffer of data. The CRC16 algorithm is used in various communication protocols to ensure data integrity.

```
//modbus_crc.c

#include "stdint.h"

/* Table of CRC values for high-order byte */

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, 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,
    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, 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
};
/* Table of CRC values for low-order byte */
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; /* high CRC byte initialized */
    uint8_t crc_lo = 0xFF; /* low CRC byte initialized */
    unsigned int i; /* will index into CRC lookup */
    /* pass through message buffer */
    while (buffer_length--) {
        i = crc_lo ^ *buffer++; /* calculate the CRC  */
        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_

uint16_t crc16(uint8_t *buffer, uint16_t buffer_length);

#endif
```

Future Enhancement

- **Web and Mobile Integration**: Develop a web dashboard and mobile app for remote monitoring and control.
- **Cloud Connectivity**: Send data to cloud services for storage, analysis, and visualization.
- **Security Enhancements**: Implement encryption and authentication for secure communication.
- **Energy Optimization**: Introduce algorithms to optimize energy consumption and reduce costs.
- **Real-Time Alerts**: Integrate notifications for critical events via SMS, email, or push notifications.
- **Enhanced UI**: Improve the user interface for easier interaction and configuration.
- **Scalability**: Support multiple devices and sensors for comprehensive monitoring.
- **Machine Learning**: Use machine learning to predict system failures and optimize operations.
- **Wireless Expansion**: Add support for protocols like Zigbee, LoRa, or Wi-Fi.
- **Power Management**: Implement battery backup to ensure continuous operation during outages.
- **Environmental Monitoring**: Integrate sensors for temperature and humidity to monitor environmental conditions.