University of Applied Sciences and Arts of Southern Switzerland
Department of Innovative Technologies

SUPSI

MSE | MASTER OF SCIENCE IN ENGINEERING

TSM_IoT - Internet of Things 2024

Students
Alessio Tommasi
Simone Vitullo
Enrico Sansonetti

Tutor
Mirko Gelsomini

Academic Year
2023 – 2024

Date
04-06-2024
(date of the defense)

Sommario

Abstract 3
Project Components and electrical schematic 3
Installation Guide 4
Code Structure and Functions 5
Main_sketch.ino 5
Model 6
SystemState.cpp 7
Destructor and Clear Function 8
Singleton Instance Management 8
State Management Functions 8
Global Declarations 8
Setup() Function 9
Destroy() Function 9
Loop() Function 9
Routes.cpp 10
DefineRoutes Function 10
Controller 11
Modbus 12
MasterModbus 12
Constructor and Initialization 12
Error Processing 12
Reading Registers 12
Task Management 12
WiFiManager.cpp 13
Connect() Function 13
ScanNetworks() 13
WiFiManager() 13
WebServer.cpp 14
Constructor 14
Begin() Function 14
NotFound() Function 14
Functions to manage error states and messages. 14
SlaveModbus 15
View 16
Conclusion 17

Abstract

The Industrial IoT project aims to integrate the ESP32 microcontroller with the Modbus protocol to create a scalable and reliable system for monitoring and controlling industrial processes. This system allows for real-time data acquisition, remote monitoring, and control through a web interface. This report provides an in-depth look into the code structure and functionality of the project.

Project Components and electrical schematic

- ESP32 Microcontroller: The core processing unit that handles data acquisition, communication, and control tasks.
- Modbus Protocol: Ensures robust communication between the ESP32 and industrial devices.
- E2002 Network Analyzer: Monitors electrical parameters, interfacing with the ESP32 via Modbus.
- Web Server and MQTT: Facilitates remote monitoring and control via a web interface.
- ESAM E2002

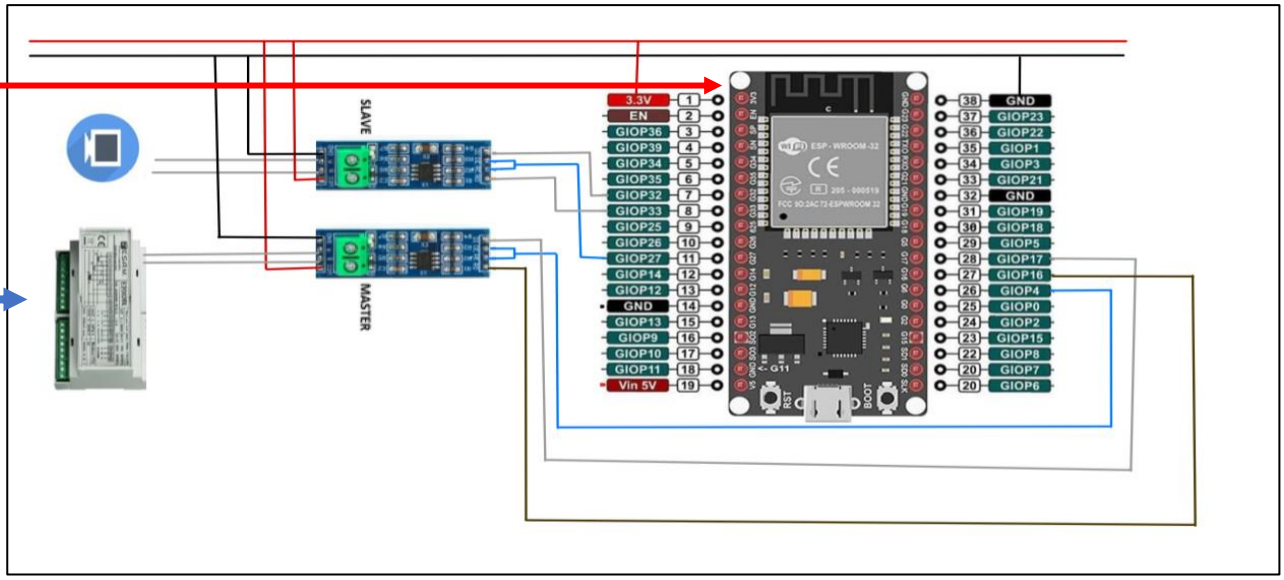


Figure 1: Electrical schematic.

The image above allows us to identify the components within our project. It also shows the physical connection of the cables that was made in our system.

Installation Guide

To compile the files in the subfolders, you need to add them as libraries (i.e., .zip files) to the Arduino IDE. Specifically, library folder to place/replace the zip files were created. To compile correctly, all zip folders must be imported into /Library:

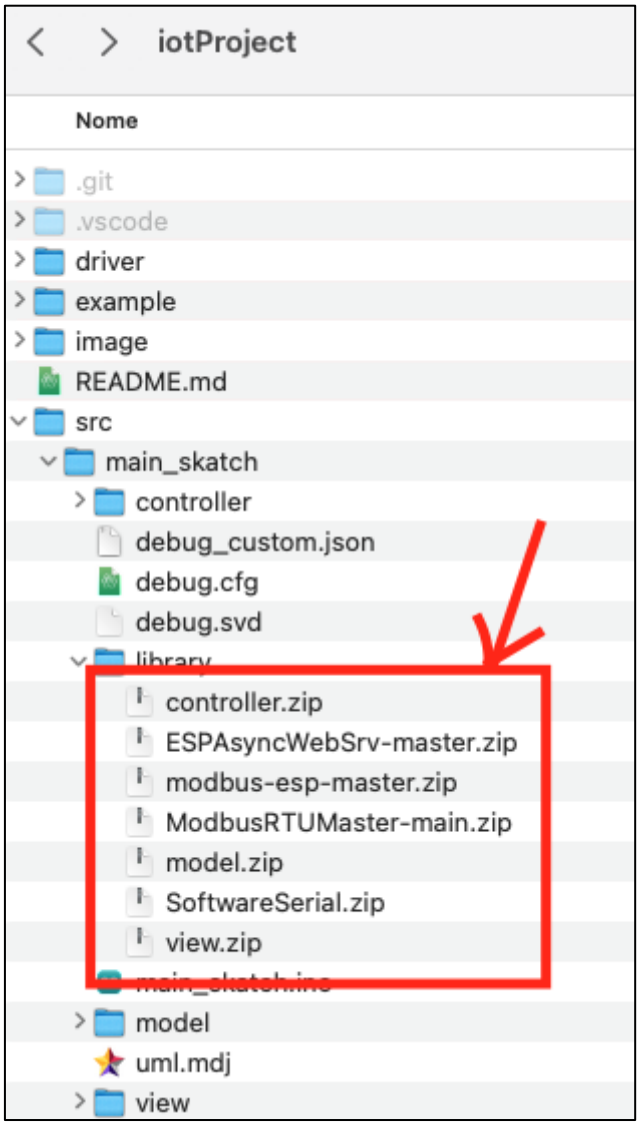


Figure 2 Dedicated library

Code Structure and Functions

The project comprises several key files, each responsible for different aspects of the system. Here we detail the primary functionalities and code snippets, explaining their roles.

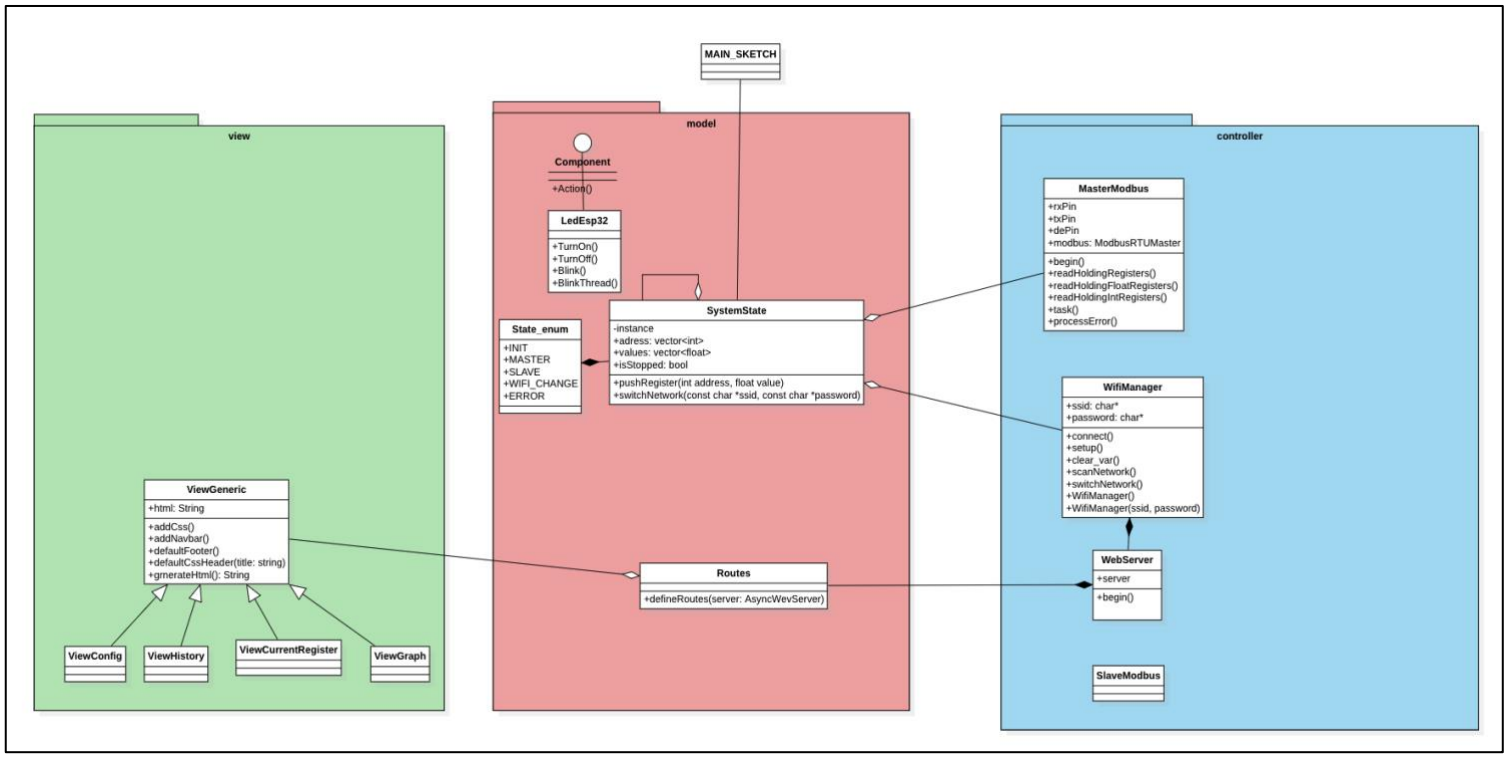


Figure 3 UML Diagram

In this following chapter, we will explain the functions contained within the classes and the structure of the code.

Main_sketch.ino

The main_sketch.ino file is the entry point for the Industrial IoT project, setting up the system, initializing components, and entering the main execution loop. This file orchestrates the integration of the ESP32 microcontroller with Modbus communication and Wi-Fi management. Below is a detailed breakdown of the functionality and key code snippets within this file.

Model

We consider this to be the folder where the main functions of the system are enclosed.

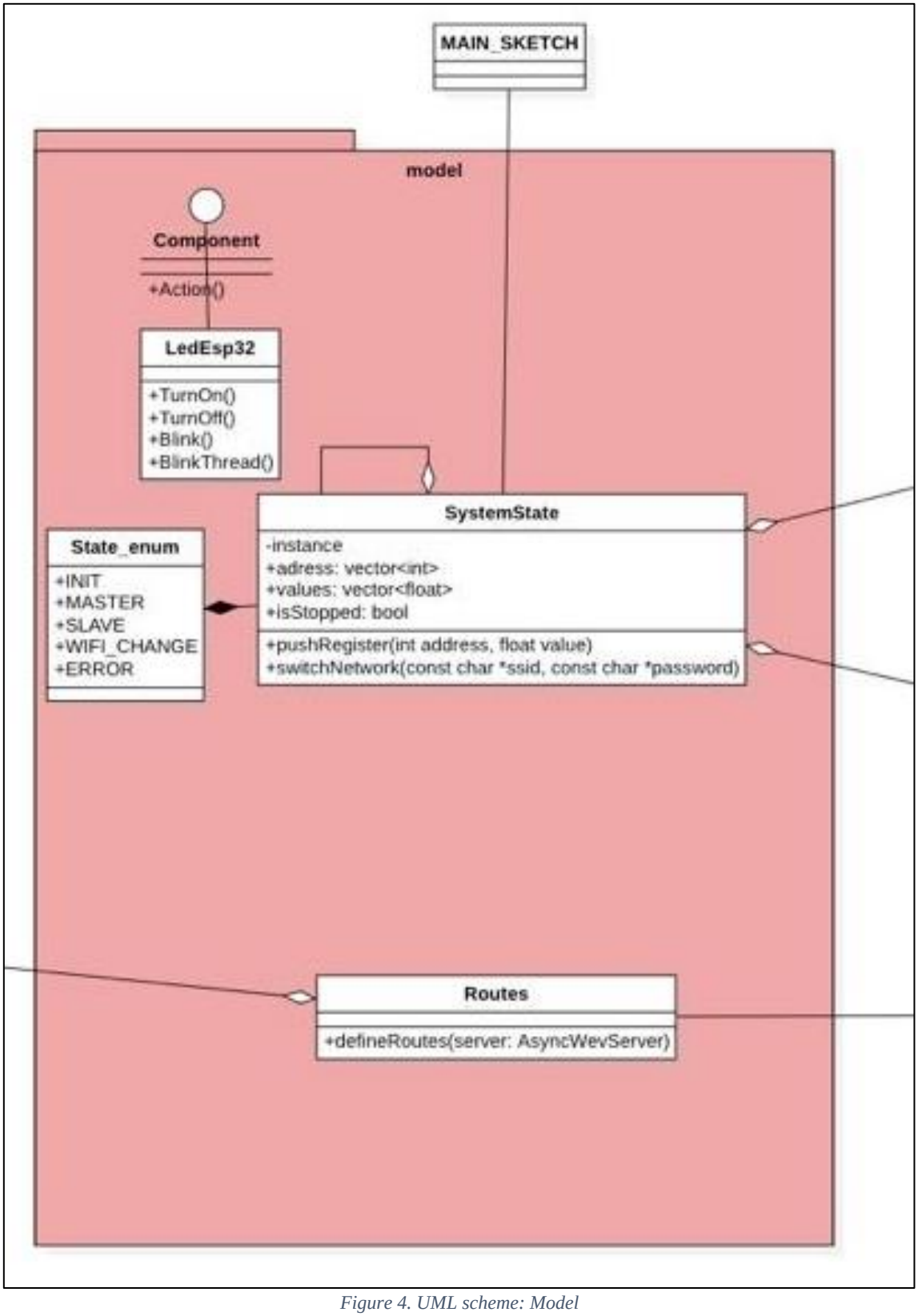


Figure 4. UML scheme: Model

SystemState.cpp

Our ESP32 has been managed and programmed as a state machine and the management of transitions between states is enabled by SystemState.cpp file. It is responsible for managing the overall state of the system, including handling Modbus communications and Wi-Fi management. This file ensures that the system can monitor and update its state dynamically, respond to errors, and handle network changes. Below is a detailed breakdown of the functionality and key code snippets within this file.

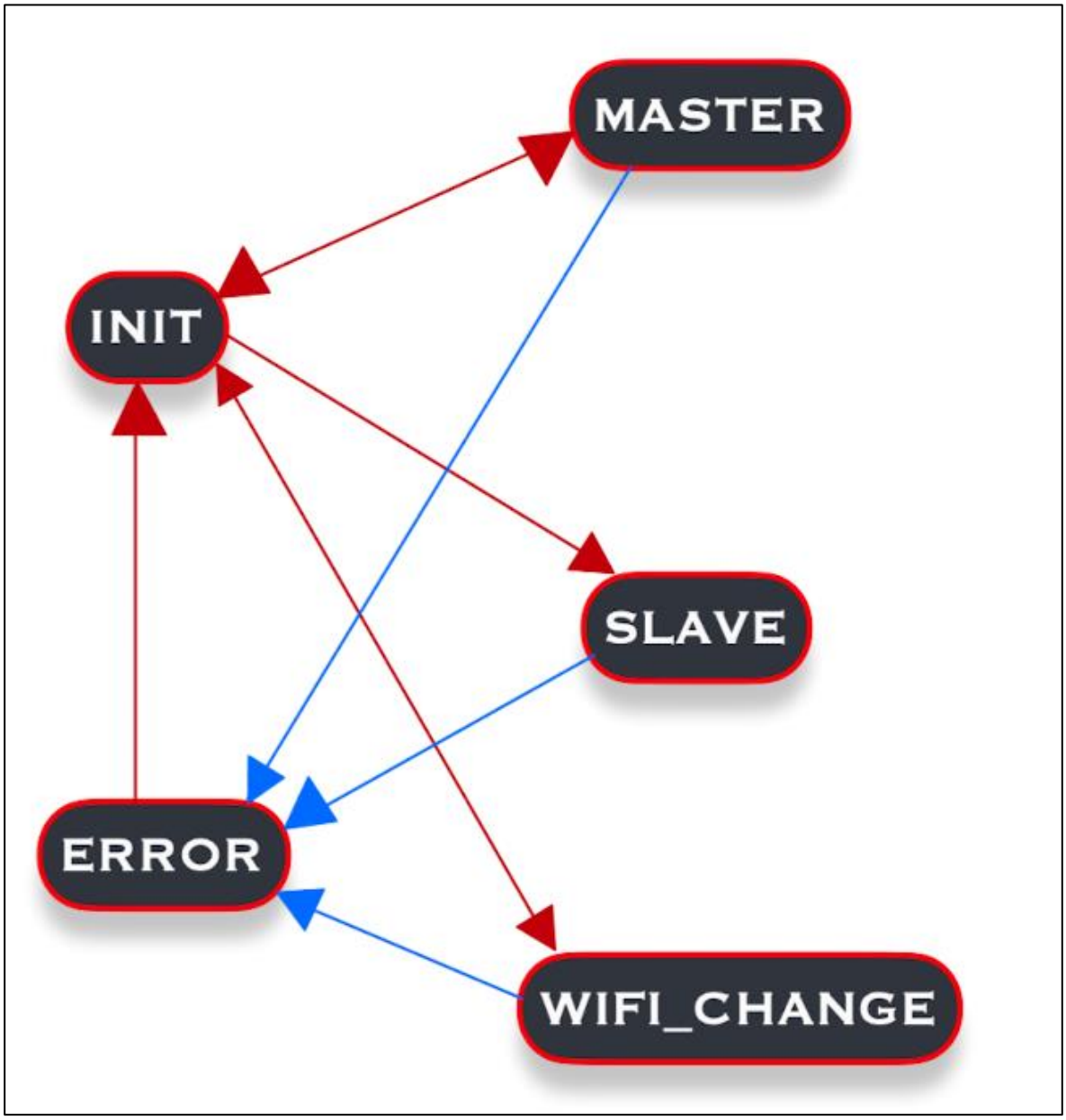


Figure 5: State machine communication diagram.

Destructor and Clear Function

The destructor and defaultClear function ensure that resources are properly deallocated.

- ~SystemState(): Destructor that calls defaultClear() to free allocated resources.
- defaultClear(): Deletes the singleton instance and masterModbus objects to ensure proper cleanup.

Singleton Instance Management

The getInstance method ensures only one instance of SystemState exists in terms of singleton pattern.

- getInstance(): Creates a new instance of SystemState and masterModbus if they don't already exist, ensuring a single instance throughout the application.

State Management Functions

These following functions handle updating and retrieving the system's state.

- update(): Placeholder for updating the system, such as modifying LEDs based on the state.
- setState(State newState): Sets the current state and calls update().
- getState(): Returns the current state.
- getStateString(): Returns a string representation of the current state.

Global Declarations

Global variables and objects are declared to manage the system state, Modbus communication, and Wi-Fi connection.

- ssid and password: Wi-Fi credentials.
- systemState: Pointer to the SystemState singleton.
- masterModbus: Pointer to the MasterModbus object.

Setup() Function

The setup() function initializes serial communication, sets up the system state, and manages Wi-Fi and Modbus initialization.

- Serial.begin(9600): Initializes serial communication at 9600 baud rate.
- systemState = SystemState::getInstance(): Initializes the system state.
- masterModbus = new MasterModbus(): Initializes the Modbus master.
- WiFiManager *wifiManager = new WiFiManager(ssid, password): Initializes the Wi-Fi manager with the provided credentials and scans for available networks.
- try-catch block: Handles exceptions that may occur during the initialization of Wi-Fi or Modbus.

Destroy() Function

The destroy() function cleans up dynamically allocated resources.

- destroy(): Deletes the masterModbus and systemState objects to free up memory.

Loop() Function

In this project it is not used and is intentionally left empty as the main operations are handled in the following files.

Routes.cpp

The Routes.cpp file is responsible for defining the routes handled by the web server which then allow the system to process different HTTP requests, enabling various functionalities like fetching data, updating configurations, and managing interactions with the user interface.

DefineRoutes Function

The defineRoutes function is responsible for setting up the various routes that the web server will respond to. It uses lambda functions to handle the HTTP requests for each defined route.

- «/»: Returns a plain text "Hello, world".
- «/history»: Generates and returns HTML content for the history view.
- «/currentregister»: Generates and returns HTML content for the current register view.
- «/getRegisterValues»: Fetches and returns register values in JSON format. Requires an address parameter.
- «/graph»: Generates and returns HTML content for a graph view.
- «/config»: Returns a simple "Config page" in HTML format.
- «/switch_wifi»: Switches the Wi-Fi network based on provided SSID and password, returning success or error messages accordingly.

Controller

This part deals with the management of wi-fi, communication via Modbus and the initialization of the web server.

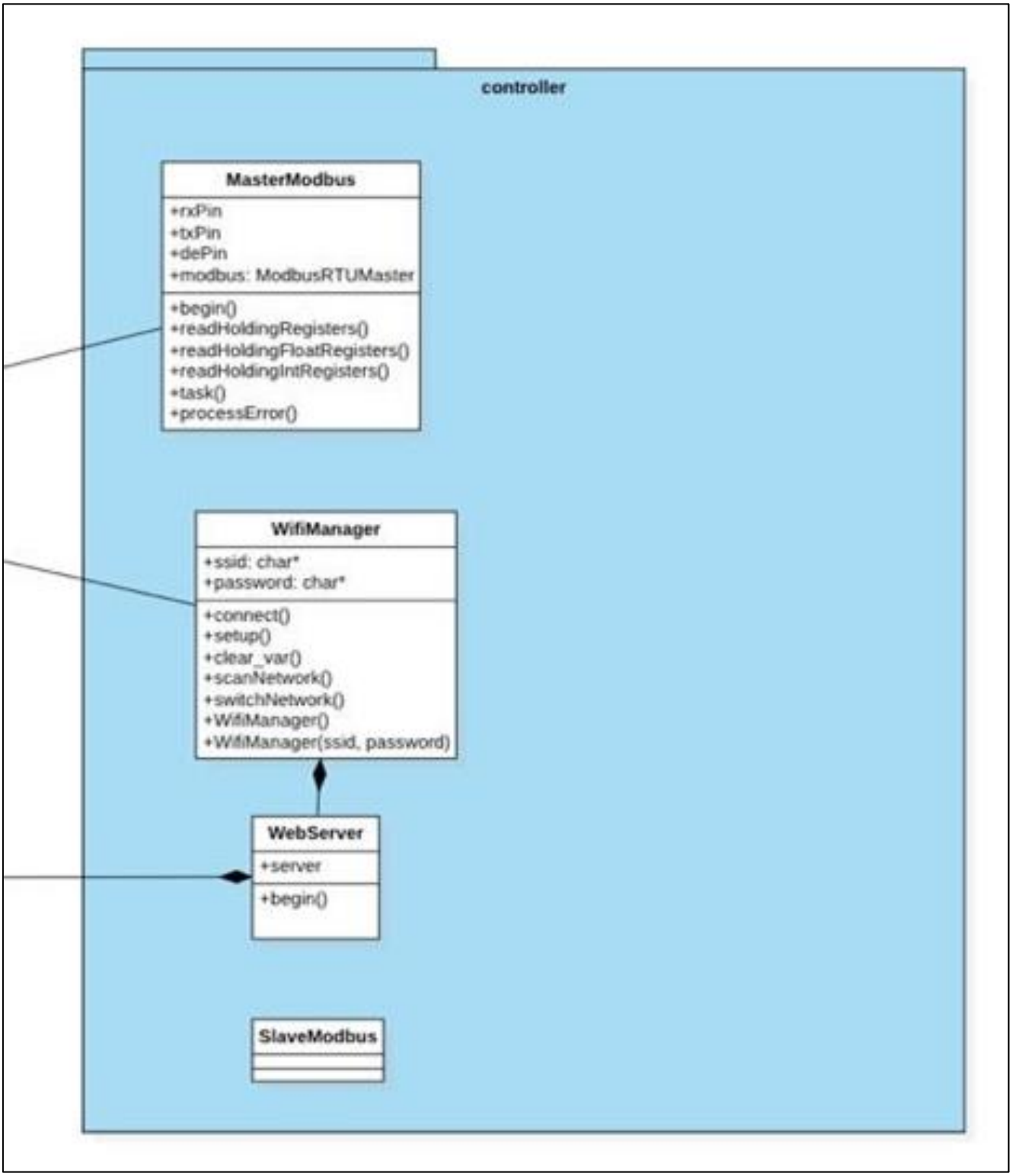


Figure 6: UML scheme: Model

Modbus

MasterModbus

The MasterModbus.cpp file defines the behavior of the Modbus master, which includes initiating communication, reading register values, and handling errors.

Constructor and Initialization

The constructor initializes the Modbus communication.

- MasterModbus(): Constructor that initializes the Modbus object and calls the begin method.
- begin(): Configures the Modbus communication parameters, including baud rate, data format, and timeout settings.

Error Processing

Handles any errors that occur during Modbus communication.

* processError(): Checks for timeout or exception responses, prints appropriate error messages, and clears the error flags.

Reading Registers

Methods to read various types of registers.

- readHoldingRegisters(): Reads 32-bit integer values from holding registers.
- readHoldingFloatRegisters(): Reads floating-point values from holding registers.
- readHoldingIntRegisters(): Reads integer values from holding registers.

Task Management

Manages periodic tasks for Modbus communication.

- task(): Reads specific holding registers as a part of periodic tasks.

WiFiManager.cpp

The WiFiManager.cpp file manages the Wi-Fi connectivity of the ESP32 microcontroller. It ensures that the ESP32 can connect to a predefined Wi-Fi network or enter access point mode for configuration in case of failure.

- The constructor «WiFiManager::WiFiManager()» initializes an ESP32 device to function as an Access Point (AP).
- WiFiManager(const char *ssid, const char *password) is the parameterized constructor that initializes the Wi-Fi credentials with provided SSID and password, and then attempts to connect.

Connect() Function

The connect() function in WiFiManager.cpp initializes the Wi-Fi in station mode and disconnects any previous connections. Afterwards the connection process using the provided SSID and password starts.

The function also continuously checks the connection status and prints the connection progress. Upon successful connection, it initializes and starts the web server. Additionally, the function handles any exceptions that occur during the connection process and rethrows them for higher-level handling.

ScanNetworks()

The setNetwork(const char *ssid_new, const char *password_new) function temporarily stores the old credentials and updates the credentials with the new SSID and password. It then attempts to connect to the new network. If the connection to the new network fails, the function restores the old credentials and rethrows the exception.

WiFiManager()

Deallocates the web server object and prints a message indicating that the WiFiManager object has been destroyed and memory released.

WebServer.cpp

The WebServer.cpp file is a crucial component of the Industrial IoT project, responsible for managing HTTP requests and serving web pages to clients. This allows users to interact with the system remotely, facilitating remote monitoring and control. The WebServer.cpp file also defines the behavior of the web server, including its initialization and how it handles client requests.

Constructor

The constructor initializes the ssid and password for the Wi-Fi connection, setting up the essential credentials for network connectivity.

- WebServer(): Initializes the WebServer object with the provided SSID and password for Wi-Fi connection

Begin() Function

The begin() sets a handler for undefined routes using the «onNotFound» method. Next is call Routes::defineRoutes(server) to define the specific routes for the web server before starting server and printing a message indicating that the server has started on serial port.

NotFound() Function

The «notFound()» function handles requests to undefined routes, returning a 404-error message.

Functions to manage error states and messages.

- setWifiManager: Sets the Wi-Fi manager instance.
- switchNetwork: Updates the state to WIFI_CHANGE and switches the Wi-Fi network using the provided SSID and password.

SlaveModbus

The SlaveModbus.cpp file defines the behavior of the Modbus slave, which includes initiating communication, managing registers, and handling Modbus tasks.

Constructor and Initialization

The constructors initialize the Modbus slave communication.

- SlaveModbus(): Default constructor that initializes the Modbus slave with a default register.
- SlaveModbus(): Constructor that sets the slave ID.
- begin(): Begins Modbus communication on the specified serial port.

Register Management

Methods to manage holding registers.

- addHoldingRegister(): Adds a holding register at the specified address.
- setHoldingRegisterValue(): Sets the value of a holding register at the specified address.

Task Management

Manages periodic tasks for Modbus communication.

- task(): Handles Modbus communication tasks.
- loopTask(): Continuously runs the task() method unless the system is stopped in async thread.

View

Each view has a generate html function inside it and via polymorphism allows me to generate the corresponding web page as reported in the images bellow.

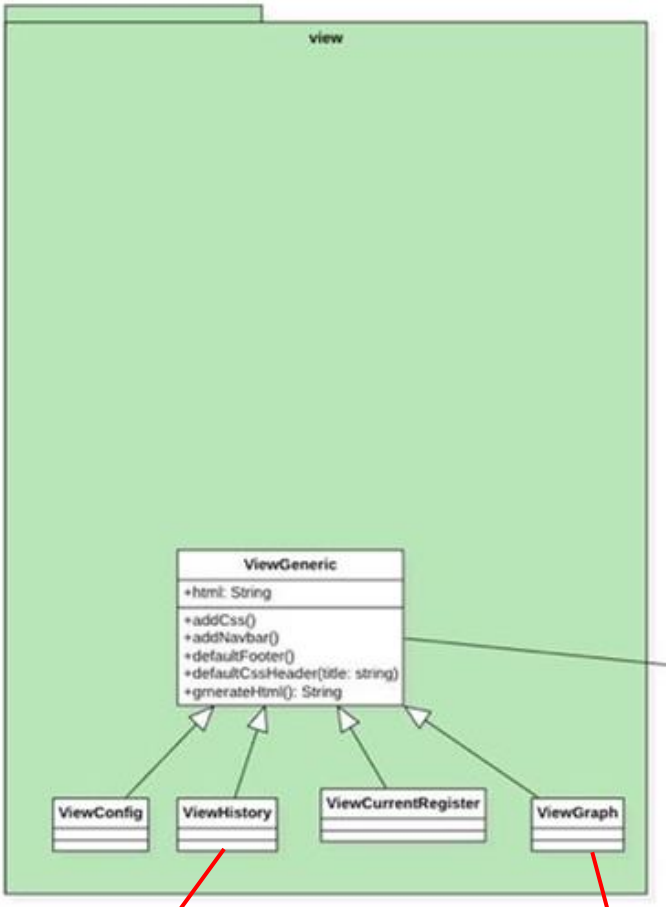
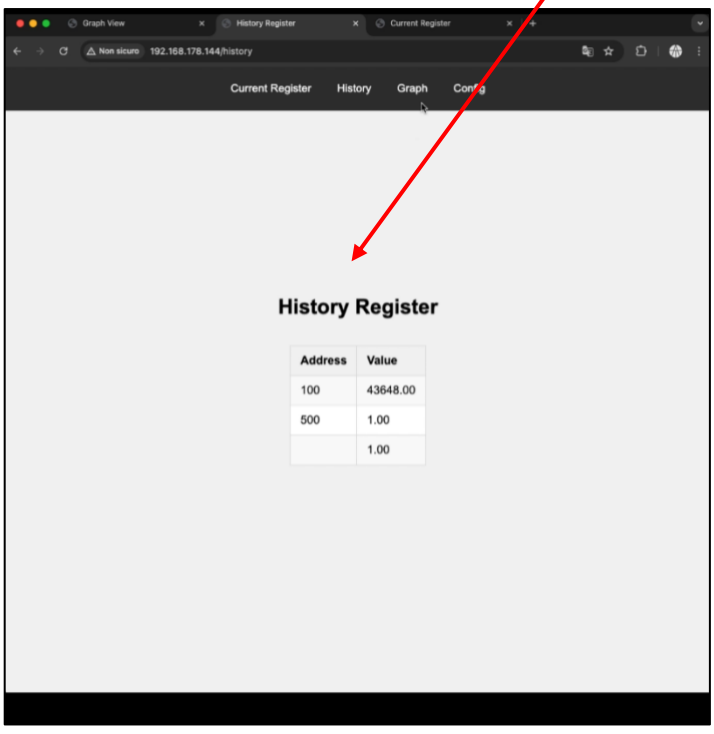


Figure 7: UML scheme: View



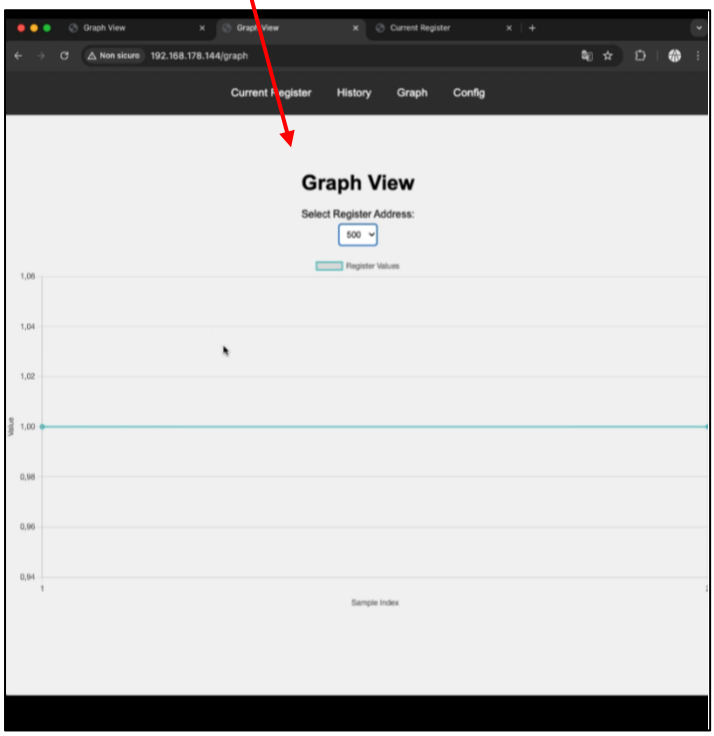


Figure 8 Html page: Historical register and Graphic view.

Conclusion

The system is able to display and connect to available networks. Communication via modbus control protocol has been implemented and this allows us to acquire and store data from our ESAM in real time. The data is displayed graphically thanks to dedicated html pages.

The Code that handles communication via PC with modscan software operating as master and ESP32 operating as slave was implemented. Currently, the functionality of this part of the code was not tested due to possible hardware problems.