Rees Raphael
Benavides Jhon
Clerici Lorenzo

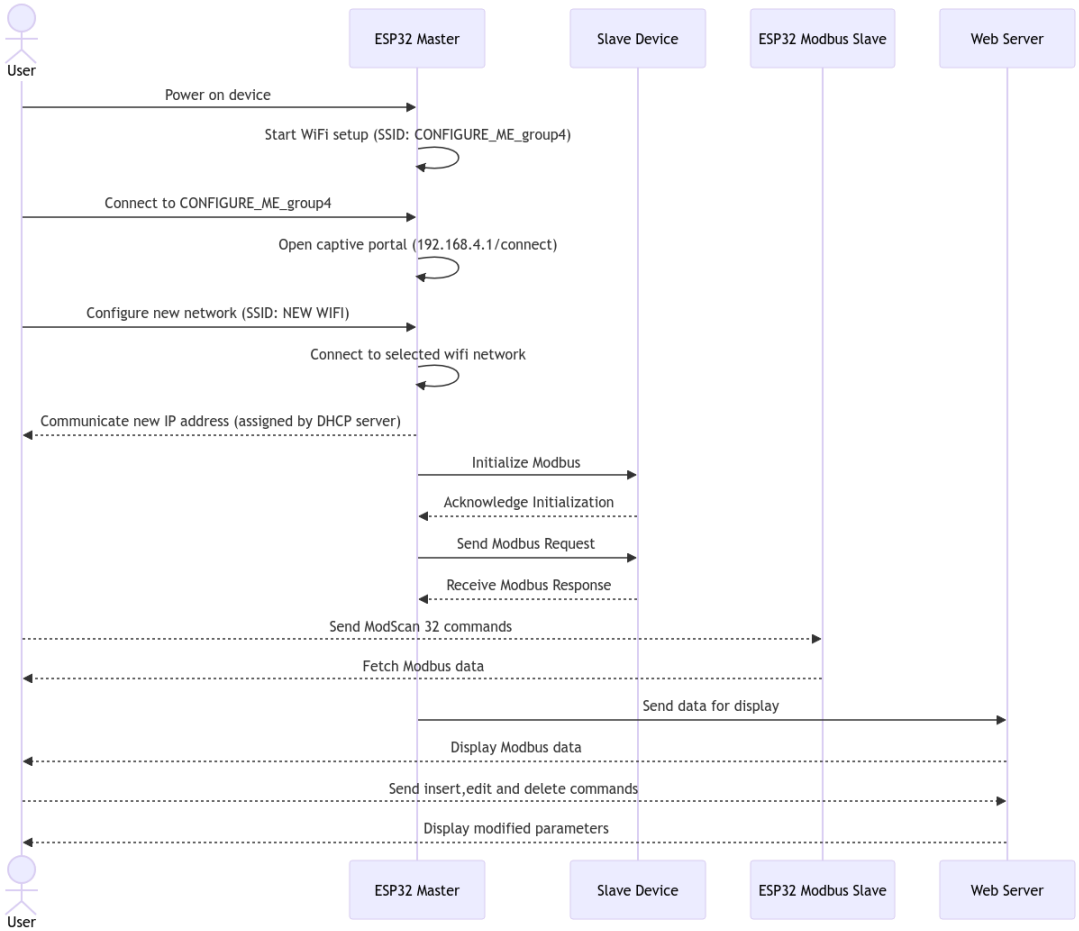
Modbus communication system with ESP32 – IOT Report (Group 4)

Introduction

The aim of this project was to develop a system capable of sending and receiving Modbus requests and displaying the data on a webserver for the user. The main components of the projects are the ESP32 microcontroller, the ESAM E2002 device, and the ModScan software. The ESP32 microcontroller can operate in two modes: master and slave. The Espressif IDF tool was chosen as the development platform due to its flexibility to work with and it leverages the possibility to work with a Real time operating system which applies load-balancing over the functionalities of this project. It also allows the programmer to change specific parameters in the future.

Overview of the System

The following diagram illustrates the overall operation of the system, detailing the interaction between the user, ESP32 Master, ESP32 Slave, Modbus Server, and Web Dashboard.



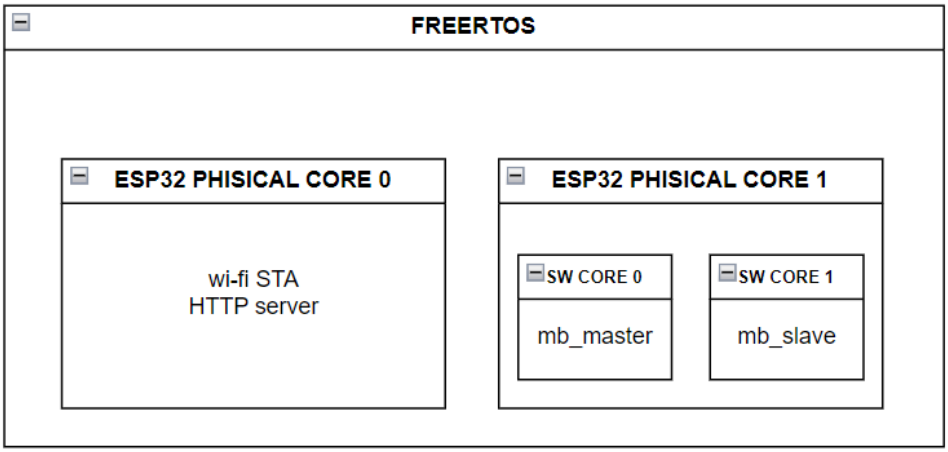
Rees Raphael
Benavides Jhon
Clerici Lorenzo

FreeRTOS:

We have opted to use FreeRTOS, an open-source real-time operating system for microcontrollers. It is designed for embedded applications that require precise and deterministic control of execution times. It is lightweight and optimized for devices with limited resources. Its main features include task management, resource synchronization, cross-platform support, and precise timing. It is widely used in industrial automation, medical devices, IoT, consumer electronics, and transportation systems. FreeRTOS offers efficiency, reliability, flexibility, and strong community support.

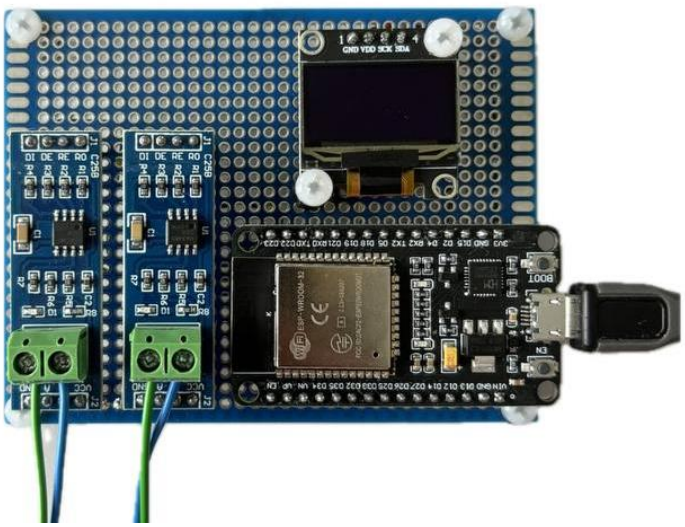
The ESP32 is equipped with two physical cores. The task allocation across the two cores is as follows:

- CORE 0: Task for managing Wi-Fi in station mode and managing the HTTP server
- CORE 1:
    - Task for managing Modbus slave
    - Task for managing Modbus master



The Hardware

we created an experimental board using a perfboard, with the aim of achieving a robust system and ensuring reliable contacts.



Rees Raphael
Benavides Jhon
Clerici Lorenzo

ESP32 – Modbus Master – Data Reception

The ESP32, operating in master mode, sends Modbus requests to the ESAM E2002 and receives data from its holding registers.

Initialization:

- During this phase the initial parameters of configuration are defined. Among them, the reception (Rx) and transmission pins (Tx) are defined. Also, the direction or RST UART pin which indicates if the ESP32 will receive or send data through the AB differential lines. Additionally, the parity and the baud rate or speed of transmission is defined, which must be the same either the master or slave Modbus devices.

Sending Requests and Receiving Data:

- The *master_operation_func* function in *master.c* is responsible for sending Modbus requests and processing the responses. This function iterates over the items of the parameter table, which contains the specification related with the data type, name, id, register address of the variable that the user wants to read. This table is available on the web interface and can be modified by the user. Subsequently, it builds the request, converts to the appropriate bit frame according to the Modbus protocol rules and sends to the ESAM E2002 fetching for the data.

- The *mbc_master_get_parameter* function also plays an important role on the system. It retrieves the parameter values from the Modbus slave device. A pointer containing the current position on the array that was defined to store the values is passed to save the current measure. The algorithm iterates over all the parameters of the table, and it can understand what data type to save in and, therefore, how to handle the response.

Data Conversion:

- The retrieved data needs to be converted. The conversion is done depending on the specification of the data as float, integer, or other data types, option present on the parameter table according to the Modbus protocol.
- The swap between the most significant and the less significant bits of the value obtained is performed by the function *swap_float_value* before storing the data on the array.

Rees Raphael
Benavides Jhon
Clerici Lorenzo

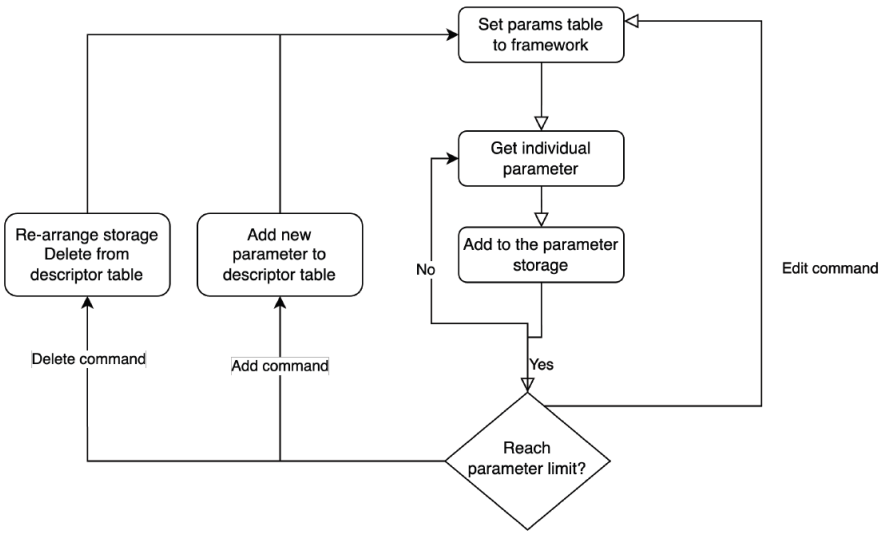
ESP32 - Modbus Master - Local Data Storage

Data Structure:

- The *holding_registers* array in *modbus_master_params.c* holds the values received from the Modbus slave device. Each entry in this array, namely the index is linked to the parameter description table by the CID column, ensuring that data is correctly matched to its data type.
- The *param_value_t* type which defines the value array in the program was created using a C union and it is used to store different types of data received from the Modbus slave.

Updating data:

- The *change_value_param_register* function updates the local storage (param_value_t array) with the new values received from the slave device.



Rees Raphael
Benavides Jhon
Clerici Lorenzo

ESP32 - Modbus Slave

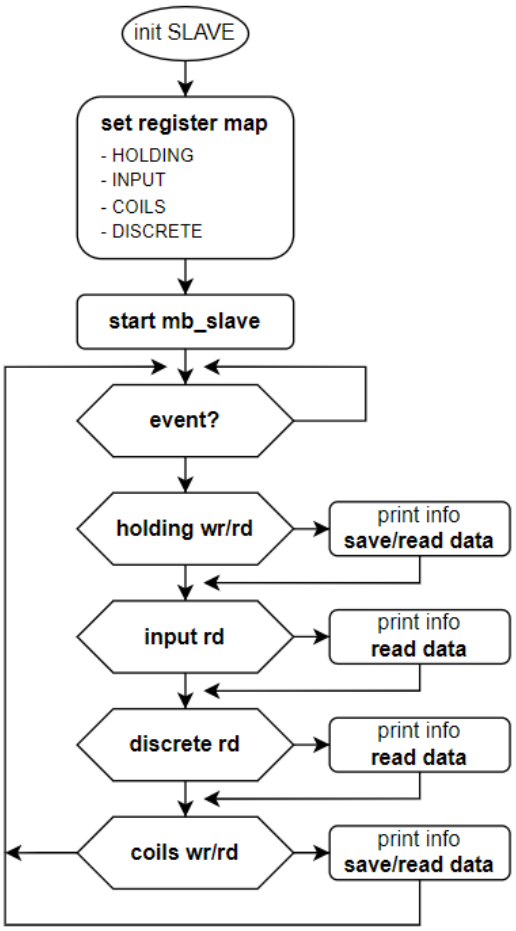
The ESP32, when operating in slave mode, responds to Modbus requests from the ModScan software. It handles requests to read and write to the holding registers.

Slave initialization:

- The *slave_init* function in *slave.c* sets up the Modbus slave by configuring the communication parameters and defining the register areas (holding registers, input registers, coils, discrete inputs).
- The *mbc_slave_init* and *mbc_slave_setup* functions are used to initialize the Modbus controller.

Handling Modbus requests:

- The *slave_loop* function runs continuously isolated into an operating system task, checking for Modbus read/write events. When an event occurs, it processes the request by reading from or writing to the appropriate register area.
- The *mbc_slave_get_param_info* function retrieves information about the Modbus parameters being accessed, and the *mbc_slave_check_event* function checks for specific Modbus events (reading/writing an holding register,input, coils, discrete ...).



Rees Raphael
Benavides Jhon
Clerici Lorenzo

WiFi Configuration

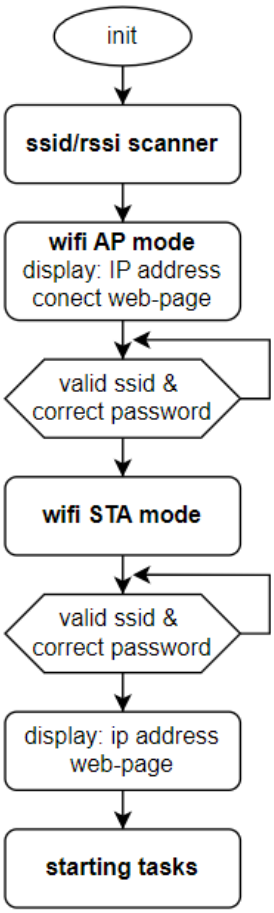
The ESP32 starts in configuration mode, creating a WiFi access point with the SSID "CONFIGURE ME_group4". A captive portal is used for configuring the ESP32 to connect to a custom WiFi network.

Captive portal setup:

- The *connect_me_operation* function in *master.c* sets up the WiFi access point and starts the web server for the captive portal. This access point allows users to connect and configure the device.
- The *wifi_init_softap* function initializes the ESP32 as a WiFi access point with the specified SSID and password.

Configuration Page:

- The *connectme.html* file provides a user interface for selecting the WiFi network and entering the password. This file includes a table displaying available networks and a form for entering the password.
- The endpoint *http://192.168.1.4/connecthttp://192.168.4.1/connect* was configured by the server to get access the network available menu.
- The function *connectme* handles the form submission, sending the selected SSID and password to the ESP32 for configuration.



Rees Raphael
Benavides Jhon
Clerici Lorenzo

IP Address Communication

Once connected to the selected WiFi network, the ESP32 communicates its new IP address to the user. The *event_handler* function in master.c handles WiFi and IP events. When the ESP32 connects to a new network and obtains an IP address, this function logs the IP address using the LCD display and updates the system state.

Configuration Page

The configuration page allows users to view and edit Modbus data entries, configure measurements, and switch between master and slave modes.

- The *index.html* file includes JavaScript functions to fetch data entries from the server, populate the table, and handle form submissions for adding or editing entries. The web page initially gets rendered empty and then the json file response from the server contains all register information.
- The *addEntry* function sends the entry data to the server, while the *editRow* function fills the form with existing entry data for editing. The *deleteRow* function handles the deletion of entries.
- The *data_get_handler, data_post_handler, and data_delete_handler* functions registered into the http server in master.c handle HTTP GET, POST, and DELETE requests, respectively, interacting with the local data storage to perform the requested operations.

Analytics Page

The analytics page displays real-time data received from the Modbus communication. This data is shown in a textual format, with values updated periodically.

- The *display.c* file includes functions to initialize the LCD display and update it with real-time data.

System stability

Functions such as *master_init* and *slave_init* in *master.c* and slave.c include error handling to log and manage initialization failures.

Conclusion

The project implements a Modbus communication system using the ESP32, capable of operating in both master and slave modes. The system allows for data reception, local storage, configuration via a web interface, and real-time data display. Future improvements could be to enhancing the user interface for better visualization and implementation of swap option directly on the user interface of the web page depending on the register selected.

Rees Raphael
Benavides Jhon
Clerici Lorenzo

Appendix 1 - physical setup of the ESP32-based Modbus communication system