Smart Street Light Control System

author Pascu Mihail-Eugen ACES 2021

1.Project Objective

The purpose of this application is to implement a system that makes the street lights more efficient than the conventional street lighting systems in the matter of **energy conservation**, **reduction of maintenance costs**, **decrease of maintenance time** and **increase in the public safety**.

2.Context of public lighting in Bucharest

The public lighting system in Bucharest is an important source of expenses for the Bucharest City Hall. [1)]According to [3] in the last 5 years, the average cost of the public lighting was around 28.650.000RON. This increased cost is due to the fact that the system was not updated to the newer and more efficient lighting solutions. **73% of the bulbs** that are used inside the lighting poles (the total numbers are around 90.000 poles and approximately 120.000 bulbs) are still **incandescent bulbs** which are less efficient and have a shorter life span than the bulbs with LEDs. The LED bulbs consume 75% less energy and have at least double life span, according to [4].
Because the civic spirit is not cultivated in our population and that no efficient maintenance system is in place for the street lighting system, **issues are detected and reported after long periods of time**. Also, because the issues are not fixed in short time, the public safety at night is at risk because of the possible street dangers (thieves, open sewer entry and others) can not be see without lighting.

3.Project Description

The system is composed of two main elements: a **central hub** and many **pole controllers**.

- ***The central hub*** can be any device that can host a web server and acts as an AccessPoint. Its purposes are:
  - to **collect diagnostic data** from the light poles;
  - **send configuration data** (time interval for turning on the light, control commands, operation modes) to the poles via the WiFi connection;
  - host the **web server** that is the user interface of the system.
- ***The pole controller*** will:
  - **detect movement**, using a PIR sensor, and tune the light intensity according to it (if movement is detected then the light intensity will be set to maximum, if not, it will be set to DIMMING_FACTOR[2)] * maximum intensity;
  - will **control the light** using different modes [3)]:
    - using the integrated environmental light intensity sensor (if it detects that is night, the lights will be turned on automatically);
    - using the time schedule interval that is provided by the central hub;

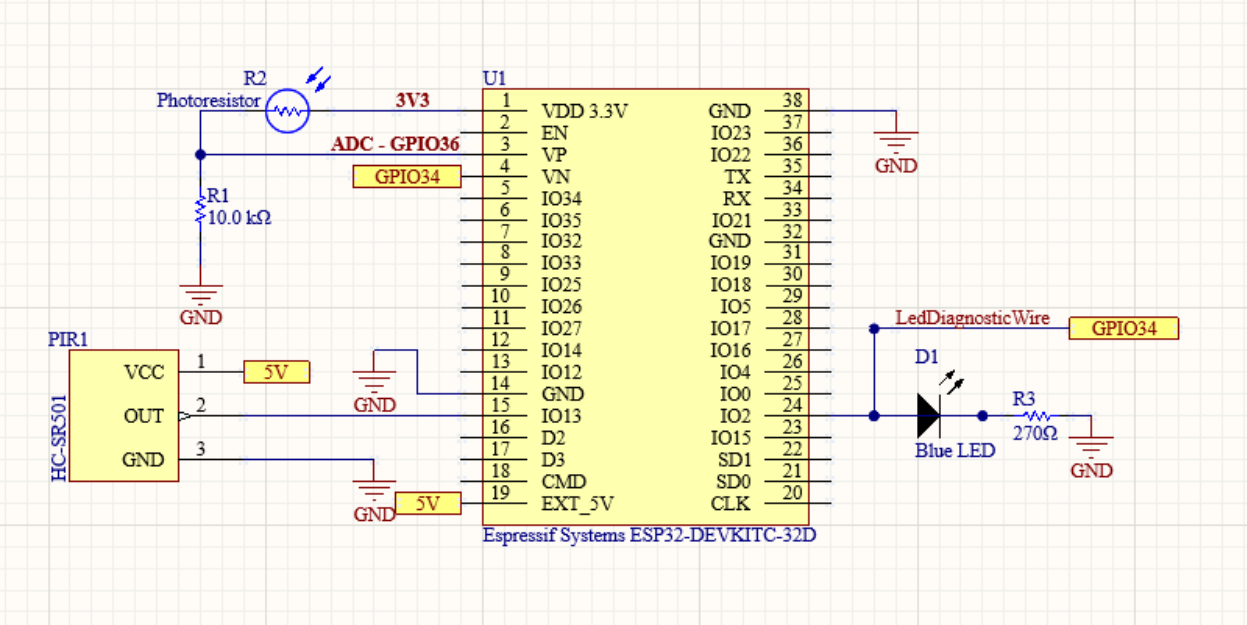
- **communicate to the adjacent pole nodes** (for extending the sensing capabilities of the poles);
- **communicate with the central hub** (for sending diagnostic data - burned bulb, missing adjacent node and others);
- **establish a mesh network** with all other poles;
- **diagnostic the LED bulb**: detect if there is a short on the bulb or if is and open circuit (missing or burned bulb);

4.Hardware Description

All the components that were used in this project are:

- **ESP32 Development board**: at least two boards are needed (one for the central hub and at least one to control the pole light). In this project I made use of four boards, one central hub and three poles. This board was chosen because of the integrated WiFi capabilities that it presents and also because it have a complex, but efficient sleep mode setting; [11];
- **Light Dependent Resistor (LDR)** or photoresistor: for each pole an LDR must be used. The purpose of this LDR is to detect the environmental light intensity, with the help of it, the pole detects if it is night or day time. Other sensors for night detection can be used, but this one had the highest efficiency/price ratio;
- **Passive InfraRed Sensor (PIR)** sensor: for each pole an PIR sensor is used to detect movement around the pole. The PIR sensor was used because it have a detection range up to 10m and a detection angle of 45degrees in each direction starting from the center [9]. These properties are in concordance with the height of the usual street poles height which is varying from 5m to around 10m [10];
- **Light Emitting Diode (LED)**: For each pole is needed one LED. This simulates the bulb of the pole;
- **Resistors**: there are used two types of resistor, both of them are needed for each present pole. The resistors are:
  - 10k Ohm Resistor: needed for reading the voltage present on LDR;
  - 15-330 Ohm Resistor: needed for current limiting the LED;

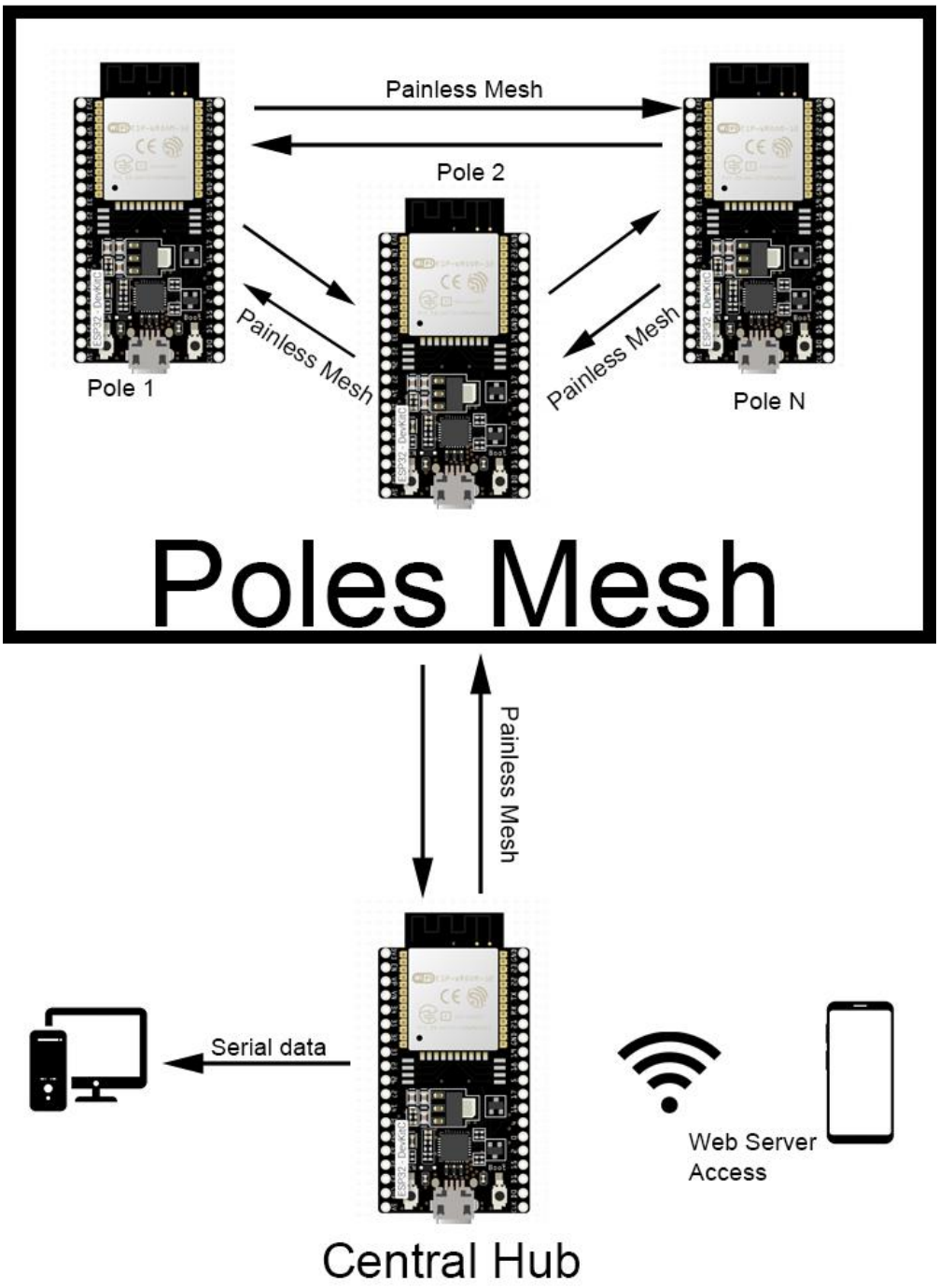
Regarding the electrical connections, the central hub has no external devices connected to it. The central hub is connected just to the PC via TTL-USB in order to communicate with the PC, to find out the IP address of the web server hosted on it. The electrical circuit for the poles is presented in the below image [5]:



In the pole schematic can be seen that one analog input (GPIO36) is used to read the value from the LDR, one analog input (GPIO 34) is used to get the voltage that the ESP32 provides to the Bulb (for diagnostic purpose), one digital input(GPIO13) is used to read the signal from the PIR movement detection sensor and one digital output(GPIO2) is used to control the bulb of the pole. The power supply issue was not taken into consideration for when the system will be installed in a pole. Currently the pole hardware is powered via USB cable. As a solution for this issue, a step-down power supply could be used to connect to the 220V network and convert to 5V, to power up the system.

5.Software Description

The system architecture can be found in the image below:

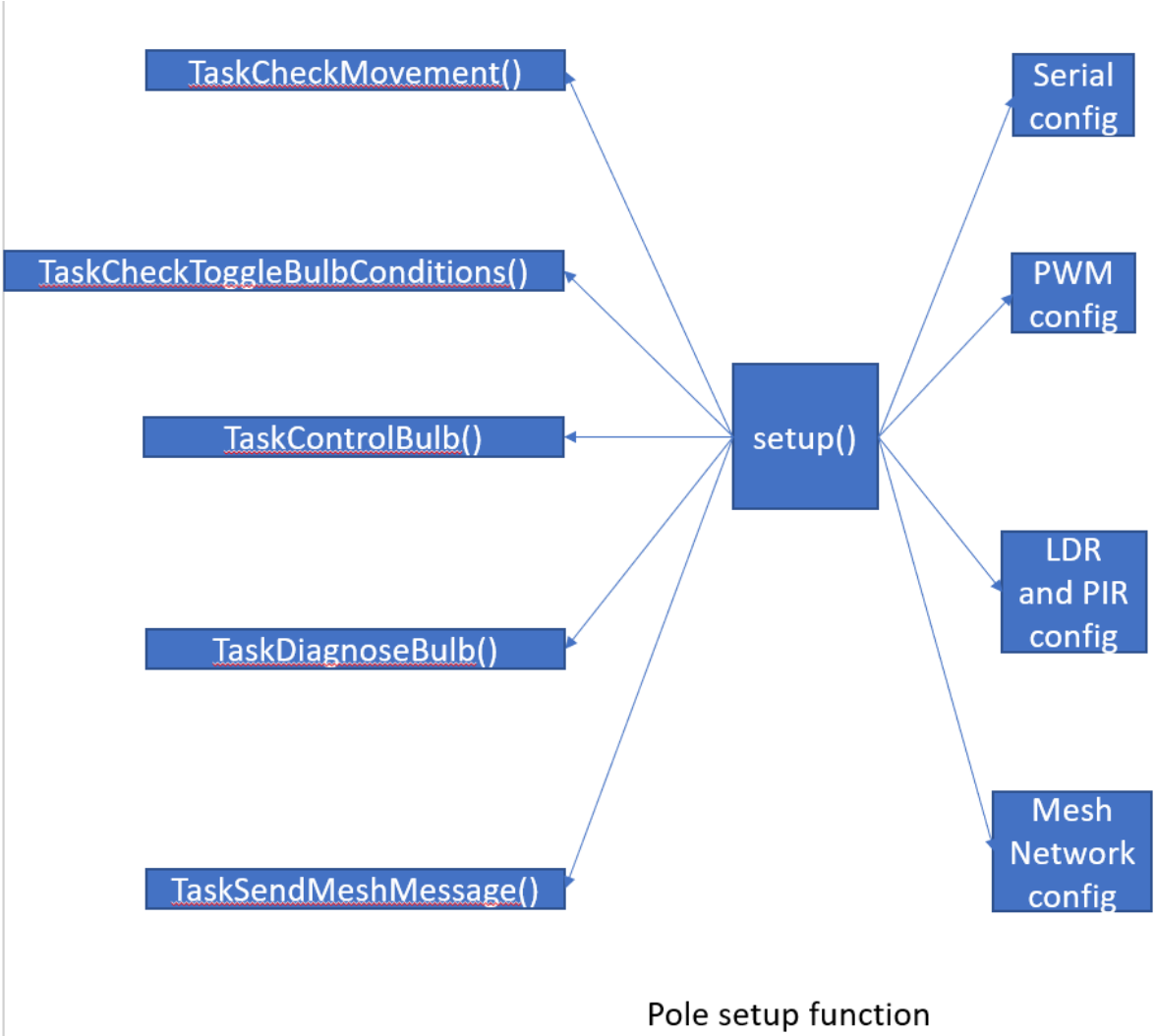


As can be seen, the poles and the central hub establish a mesh network making use of the library painlessMesh [6][7]. This handles the message transmission environment between the poles and the centralHub. Beside this part of **code that handles the mesh**, there is also the **logic part** for handling the poles functionalities. This part is making use of the FreeRTOS [8] concepts, cyclical tasks.
Also, the software that was developed for this project is divided in two components:**Pole** and **CentralHub** code. Each part was again divided in multiple files to improve the code readability, update and maintenance time in this way (<part> can be pole or centralHub):

- <part>Code - file that contains the preprocessor directives (defines, includes), the new types defined for this project, the external used variables, the declaration of the functions and tasks that will be used and the definitions for the setup and loop functions;
- globalParameters - global parameters for the <part>;
- meshFunction - contains the definitions of all the functions that are used in the mesh network communication, maintenance and event processors;
- <part>Functions - contains the definitions of all the functions that handles the logic for the applications of the <part>;
- tasks - contains the definitions for the tasks that are used to call cyclically the logic handling part of the code;

The software implemented make use of the following libraries: **WiFi, SPIFFS, painlessMesh and ESPAsyncWebServer**.

Pole Code is composed of the setup and loop functions and 5 tasks. This can be seen in the below diagram.



Pole setup function

A detailed explanation regarding the functionality for each part is presented in the next rows.

- **setup function** configures the tasks that will be runned by the scheduler of the FreeRTOS, the serial console (used to get the status of the CentralHub that is connected via USB to a PC), the PWM for the Bulb and the pins that are used as input for the Photoresistor and PIR MovementDetection Sensor;
- **loop function** is used to call the function update() from the painless mesh API. It's purpose is to maintain the mesh network active;
- **TaskCheckMovement** is used to check cyclical if there is an positive detection from the PIR Movement Detector and update the global parameter of the pole that stores this value;
- **TaskCheckToggleBulbConditions** is used to check cyclical if the mode of Bulb Control was changed. This can be changed by the user using the web server from the centralHub. The two modes control the lights by using a predifined time interval of by detecting if the environmental light intensity is below a certain threshold.
- **TaskControlBulb** [15] has the sole purpose to control the PWM of the Bulb based on the conditions of function for the light. If the turn on conditions are not fulfilled the bulb will

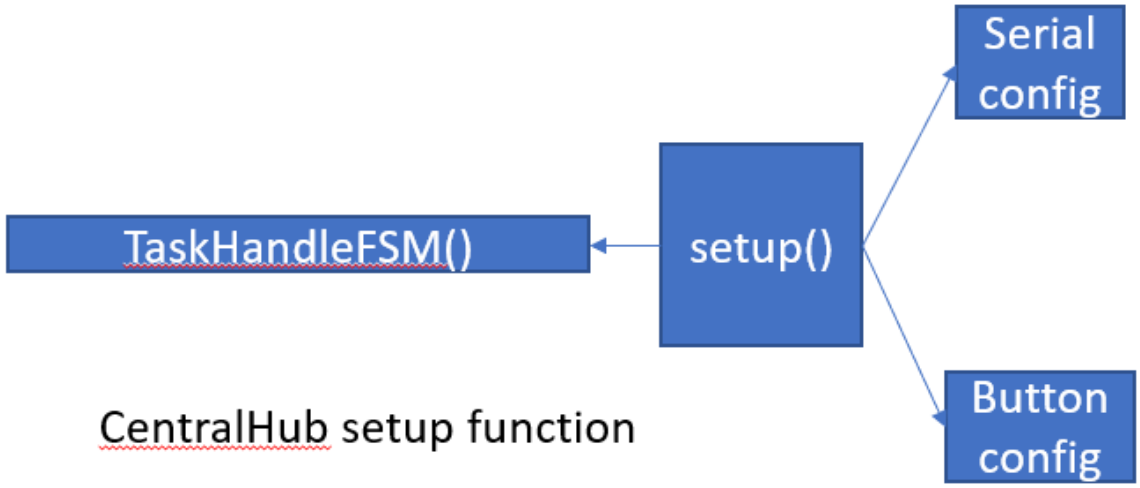
stay off. Else, if movement is detected it sets the maximum PWM cycle. If no movement is detected, after 4 minutes, the bulb will be set to minimum intensity.

- **TaskDiagnoseBulb** [12][13][14][16] is the task that checks the analog read from the input of the LED to detect what is the status of the Bulb. This task is executed 100x slower than the other tasks of the pole because it occupies a lot of CPU cycles. This intensive used of CPU cycles is the result of an implementation for robust reading of the PWM output for the LED. If one read was made, it could be on the low side of the period of the PWM. That is why, to prevent this, multiple reads (by experiment 10 are chosen) are made on a single PWM Period and if a signal greater than 0 is detected, then we are in a high side of the PWM Cycle. The states that the simple circuit can detect are:
    - open circuit - value read is greater that a maximum threshold (by experiment it was detected to be around 100 units below the maximum level <4095> of the ADC);
    - shorted - value read is smaller than a minimum threshold (by experiment it was detected to be around 1000 unit of the ADC);
    - turned off - value read is 0;
    - minimum intensity - value read is maximum intensity divided by the dimming factor (chosen by experiment to 5) and conditions to turn on Bulb are true;
    - maximum intensity - value read is maximum intensity (4095) and conditions to turn on Bulb are true;
- **TaskSendMeshMessage** is used to send cyclical to the centralHub a message containing an JSON object with the poleID, statusOfBulb, bulbLightIntensity, nightDetected and movementDetected information from the pole that sends the message. Because of the use of the JSON object, the internal stack allocated for this task was increased 4 times compared to the other common tasks of the pole.

Beside these tasks, on event trigger is used in the pole code. This is used when a new message is received from the centralHub. This message is in JSON format and it contains the mode in which the pole will make the control of the Bulb (by light intensity of by time interval).

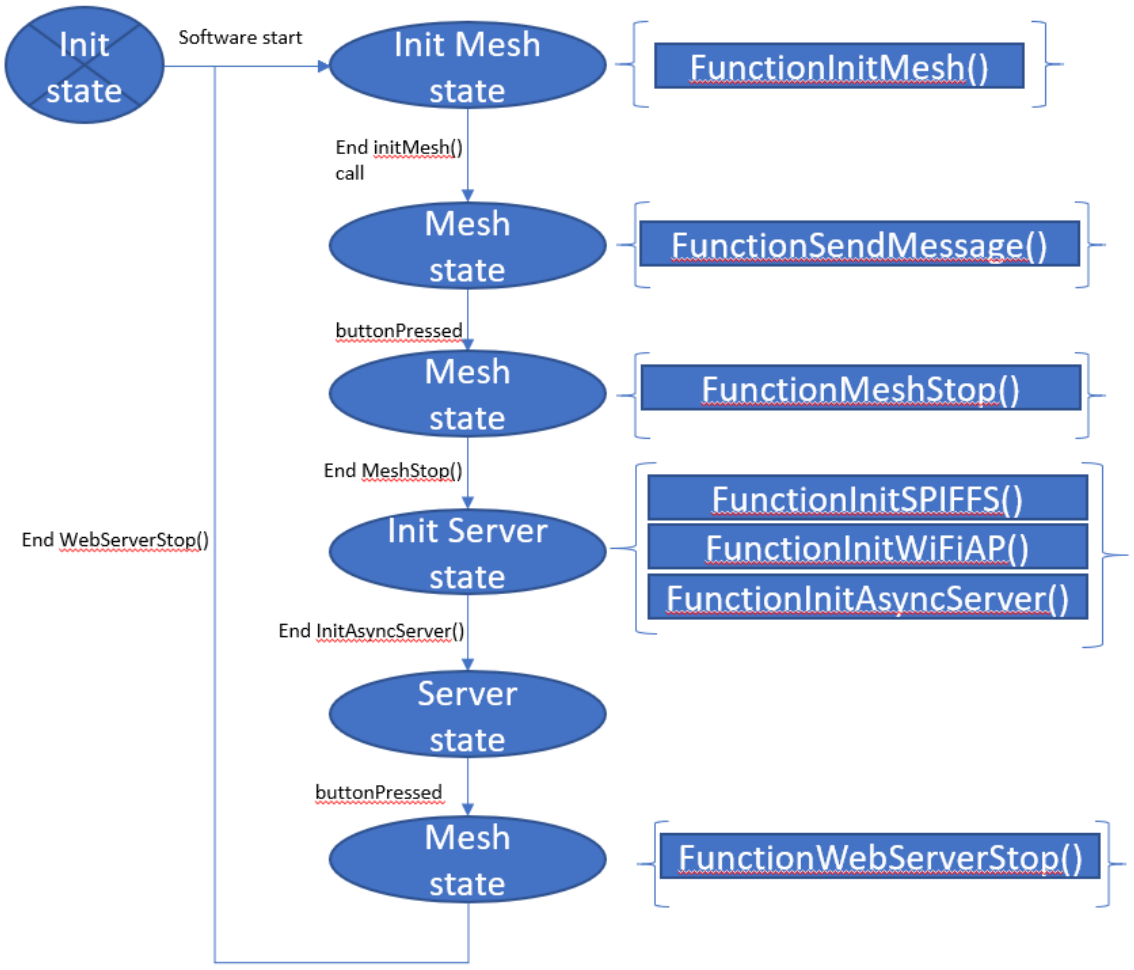
Central Hub code is splitted in two parts: the code that runs on the ESP32 and handles the mesh network connection and the functionalities implementation (written in C); and another part that handles the web server part (written in HTML,CSS and JavaScript) [17][18].

- **The C part** of code is composed of setup and loop function, one cyclic task, multiple event triggered functions and one interrupt service routine(ISR). This part implements a finite state machine that is used to toggle between the webServer mode and the mesh network communication.



CentralHub setup function

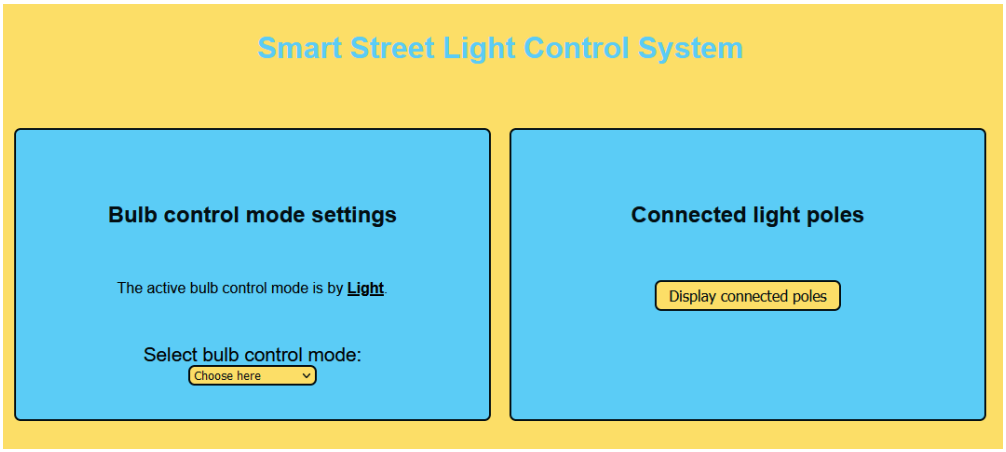
CentralHub HandleFSM Task

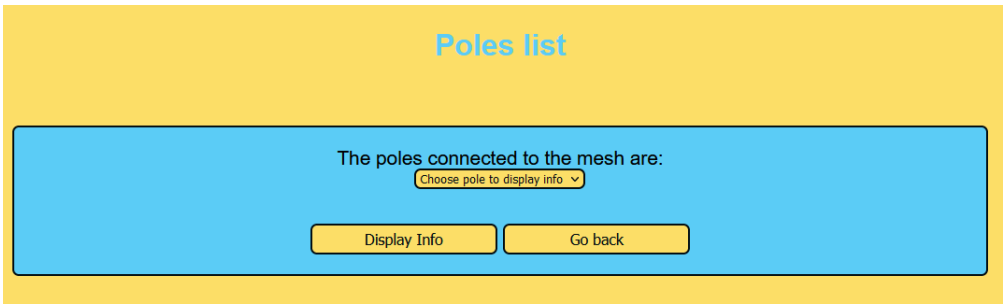


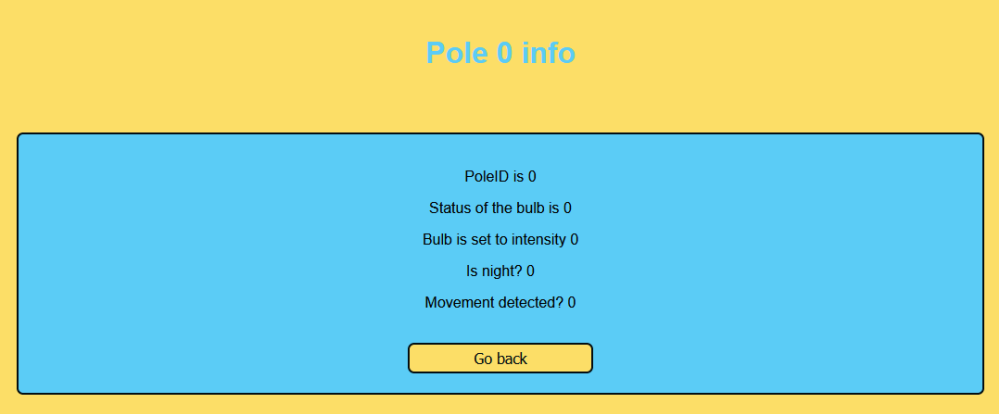
The detailed explanation for each of the components of the C part code is:

- **setup function** handles the configuration for the ISR and the control button of the FSM and also the task TaskHandleFSM configuration;
- **loop function** triggers the mesh.update() function only if the FSM is in the meshState;
- **TaskHandleFSM** handles the FSM that is responsible for the behavior of the centralHub. This is the hearth and the brain of the centralHub. It was implemented because, between the AsyncWebServer and the painlessMesh, was present a conflict in using the WiFi dedicated hardware. So a split, in time, was made between the two. When one of them is active, the other one is automatically deactivated by this FSM;
- **FunctionReceivedCallback** handles the JSON response from each of the pole and store the data from them to a global array of poles.
- **ISR** is used to detect if the button on board was pressed;

- **The WebServer** part is composed of 5 files, 3 to handle the web pages, one for css style and one of JavaScript for dynamically populating the list of available poles from the array stored internally with the C implementation. The HTML pages are:







- a start page from which you can configure the control mode for the poles and to display the list of connected poles;
- a polesList page that displays the list with all the connected poles;
- a poleInfo page that displays the data for a selected pole (poleId, status of the bulb, bulb intensity, if is detected night and if is detected movement);

6.Issues and Solutions

During the development of this project, series of obstacles and blocking points were found and overcome. Mostly were due to the fact that no prior experience was present with the ESP32 and the mesh networks. But with a deep research, all the issues were overcome. A list with a part of the issue found and solved:

- When the architecture of the system was made, the central hub was designed to communicate with the poles, but also with the device that the user is using to access the WebServer of the ESP32. This lead to the problem that both, the mesh network and the configuration as AP to make the WebServer accessible to user, were using the same WiFi hardware. This lead to issues like Core_Dump and WatchDog resets. The **ideal solution**, that I came up with, was to use another intermediary board. In this way, the central hub is only AP and host of the WebServer. It communicates serial with the intermediary board which collects data from the poles and sends it to the centralHub via that serial communication. This ideal solution was not implemented because of time constraints, but the **solution with the FSM** was the one that solved the problem within the limited time;
- Bulb state diagnostic (burned, not connected or normal operation) was another field in which I was not prepared at all, but with the needed research, the problem was rapidly passed through;
- A difficult issue was also the fact that, after reading the PWM value that was transmitted to the LED for the diagnostic purpose, the value returned by the ADC appeared to be random. After finding out how PWM works, the solution came: decrease the PWM frequency and make multiple reads for the value that it returns.
- Regarding the detection of the movement, the implementation for the delay for the bulb to stay on for a period of time after movement was detected, was another stepping stone. A timer in the FreeRTOS could be implemented, but after reading the PIR sensor datasheet [9], I saw that, by adjusting an potentiometer, the delay time in which the signal

OUT stays HIGH can be varied from instantly to even 4 minutes, which for this application is exactly what is needed;

7.Conclusions

The purpose of this project was to create a prototype that would show alternatives and improved ways of how the public lighting system could be improved. There is a long way until the classic lighting system will be replaced by smart one and limitations (like initial cost investment, cost of development and effort of installation), but, as shown in this project, a smart public lighting solution could solve many of the current issues. So maintenance time and cost will be decreased, power consumption will be decreased and public safety will increase.

The lessons learned during the development of this project are not few and here are just some of them:

- how an RTOS is working;
- the importance of a good system architecture before the starting of the software development;
- the EPS32 microcontroller;
- web server development (html, css and javascript);

For future development, a list of improvements will be provided:

- added geolocation (using a GPS module) of the pole: in this way, when the pole report a faulty bulb, it will send also the location of it so the maintenance team will fix the issue as soon as possible. A simpler and economical implementation (not using GPS module) is to give the poles some naming related to their location;
- the web server to run in parallel with the mesh network: in this way all the configuration changes requests from the user will be done instantly to the poles; and also, the data from the poles will be refreshed on the web server as soon as available;
- make use of the sleep modes of the ESP32 to reduce even further the power consumption of the system;
- light to be controller via a fixed time interval;
- the central hub could log data to a cloud, for it to be available to the user any time it has internet connection;
- the poles to communicate with adjacent ones for extending the sensing capabilities (movement, light intensity, others) of the poles;
- the IP of the webServer to be displayed on a screen that is connected to the centralHub in order to avoid the serial connection between the computer and the centralHub. In this way, the system will me more independent from other devices and also, it will be easier to use.

8.Bibliography and Resources

[1] - Verison Intelligent Lighting Solutions :Smart Street Lighting System
[2] - iNELS SMART CITY - Smart street lighting

[3] - Costul anual al iluminatului public in Bucuresti
[4] - Incandescent vs CFL vs LED bulbs
[5] - NodeMCU ESP32 Pinout
[6] - PainlessMesh
[7] - PainlessMesh API
[8] - FreeRTOS
[9] - PIR Datasheet
[10] - Street Light Pole Height
[11] - ESP32 SleepModes
[12] - LED Broken Values
[13] - LED sense circuit
[14] - How LEDs Fail
[15] - ESP32 PWM
[16] - Read PWM value with ADC
[17] - ESP32 Asynchronous Web Server
[18] - ESP32 Web Server using SPIFFS

Download the source code

pascumihaileugen_aces_iotproject.rar

[1)] CMIPB - A non official source of information was used, because the CMIPB - Company of Public Lighting Bucharest site is not available for official data.
[2)] Parameter that is configurable via the central hub
[3)] More modes could be added in next iteration steps