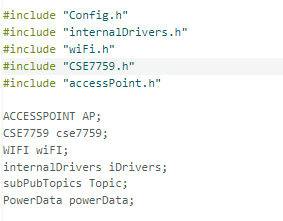
**Initialization of Modules and Global Instances**

****

1. **Header Files**: Includes configuration and module headers for Wi-Fi, access point, energy metering (CSE7759), and internal drivers.
2. **Object Instantiation**: Creates instances of classes for managing:
   * AP: Handles access point mode.
   * cse7759: Interfaces with the CSE7759 energy metering IC.
   * wiFI: Manages Wi-Fi connectivity.
   * iDrivers: Controls low-level internal drivers.
   * Topic: Manages MQTT or messaging topics.
   * powerData: Stores power measurement data.

**Start of setup()**

**iDrivers.gpioInit() :-**

This function, gpioInit, is a member function of the internalDrivers class and is responsible for initializing specific GPIO.

1. pinMode(\_relayPin, OUTPUT); :
   * Sets \_relayPin as an output to control a relay.
2. pinMode(motionDetectionLed, OUTPUT); :
   * Sets motionDetectionLed as an output for an LED that indicates motion status.
3. pinMode(\_PIRSensorPin, INPUT);:
   * Sets \_PIRSensorPin as an input to read signals from a PIR motion sensor.





**iDrivers.rtcInit() :-**

* This function initializes the RTC. If the RTC isn’t detected or has lost power, it resets the time to the code’s compile date.

**iDrivers.readDataFromEEPROM()**

1. EEPROM.begin(512);:
   * Initializes the EEPROM with 512 bytes of storage.
2. espRestartFlag = EEPROM.read(espRestartFlagEEPRMAddr);:
   * Reads and stores the ESP restart flag from the specified EEPROM address.
3. relayControl(bool(EEPROM.read(loadStateEEPROMAddr)));:
   * Reads the relay state from EEPROM and sets the relay control based on its value.
4. ssidLength = EEPROM.read(ssidLenghtEEPROMAdd);:
   * Reads the length of the SSID (Wi-Fi name) from EEPROM.
5. Serial.print("ssidLength : "); / Serial.println(ssidLength);:
   * Prints the SSID length to the Serial Monitor for debugging.
6. passwordLength = EEPROM.read(passwordLenghtEEPROMAdd);:
   * Reads the length of the Wi-Fi password from EEPROM.
7. SSID = loadStringFromEEPROM(ssidEEPROMAdd, ssidLength);:
   * Loads the SSID string from EEPROM based on its length.
8. PASSWORD = loadStringFromEEPROM(passEEPROMAdd, passwordLength);:
   * Loads the Wi-Fi password from EEPROM based on its length.
9. autoMotionDetect\_Flag = EEPROM.read(autoMotionFlagEEPROMAddr);:
   * Reads the auto-motion detection flag from EEPROM.
10. iiiiDrivers.fetchScheduledTimes();:
    * Calls a function to fetch the scheduled times, possibly from EEPROM.





**iDrivers.readDataFromEEPROM() :-**

1. EEPROM.begin(512);:
   * Initializes the EEPROM with 512 bytes of storage.
2. espRestartFlag = EEPROM.read(espRestartFlagEEPROMAddr);:
   * Reads and stores the ESP restart flag from the specified EEPROM address.
3. relayControl(bool(EEPROM.read(loadStateEEPROMAddr)));:
   * Reads the relay state from EEPROM and sets the relay control based on its value.
4. ssidLength = EEPROM.read(ssidLenghtEEPROMAdd);:
   * Reads the length of the SSID (Wi-Fi name) from EEPROM.
5. passwordLength = EEPROM.read(passwordLenghtEEPROMAdd);:
   * Reads the length of the Wi-Fi password from EEPROM.
6. SSID = loadStringFromEEPROM(ssidEEPROMAdd, ssidLength);:
   * Loads the SSID string from EEPROM based on its length.
7. PASSWORD = loadStringFromEEPROM(passEEPROMAdd, passwordLength);:
   * Loads the Wi-Fi password from EEPROM based on its length.
8. autoMotionDetect\_Flag = EEPROM.read(autoMotionFlagEEPROMAddr);:
   * Reads the auto-motion detection flag from EEPROM.
9. iiiiDrivers.fetchScheduledTimes();:
   * Calls a function to fetch the scheduled times, possibly from EEPROM.



**if (!autoMotionDetect\_Flag) {**

**digitalWrite(motionDetectionLed, LOW);**

**} :-**

* First, we readtheautoMotionDetect\_FlagfromEEPROM.
* Then, based on the value stored in EEPROM (0 or 1), it decides whether to turn off the motion detection LED LOW or HIGH

**apSSID = wiFI.prepareDevID(MAC, apSSID) :-**

1. String WIFI::prepareDevID(byte mac[], String devPref):
   * Defines a function that takes a byte array mac[] (MAC address) and a string devPref (device prefix) as inputs.
2. char devID[30];:
   * Declares a character array devID to hold the formatted device ID string.
3. snprintf(devID, sizeof(devID), "%s%02X%02X%02X", devPref.c\_str(), mac[3], mac[4], mac[5]);:
   * Formats the devPref and the last three bytes of the MAC address into the devID array as a string in the form of a prefix followed by the hexadecimal representation of the MAC bytes.
4. return String(devID);:
   * Converts the devID character array to a String and returns it.





**deviceId = wiFI.prepareDevID(MAC,devNamePrefix):-**

1. deviceId = wiFI.prepareDevID(MAC, devNamePrefix);:
   * Calls the prepareDevID function of the WIFI class to generate a device ID. It passes the MAC address array and the devNamePrefix string as parameters, and the result is stored in deviceId.
2. String WIFI::prepareDevID(byte mac[], String devPref):
   * This function definition indicates that it takes a byte array mac[] (MAC address) and a String devPref (device prefix) as arguments.
3. char devID[30];:
   * Declares a char array devID with a size of 30 to store the formatted device ID.
4. snprintf(devID, sizeof(devID), "%s%02X%02X%02X", devPref.c\_str(), mac[3], mac[4], mac[5]);:
   * Uses snprintf to format the prefix (devPref) and the last three bytes of the MAC address (mac[3], mac[4], mac[5]) as two-digit hexadecimal values and stores the result in the devID array.
5. return String(devID);:
   * Converts the devID character array into a String and returns it.



**AP.accessPointSetup()** :-

1. if (((SSID == "") && (PASSWORD == "")) || ((ssidLength == 255) || (passwordLength == 255))):
   * Checks if the SSID or PASSWORD are empty, or if the ssidLength or passwordLength values are set to 255 (indicating no valid data).
2. readSsidAndPasswordFromAP(apSSID.c\_str(), apPassword.c\_str());:
   * If the condition is true, it reads the SSID and password from an access point using apSSID and apPassword.
3. while (((SSID == "") && (PASSWORD == "")) || ((ssidLength == 255) || (passwordLength == 255))):
   * A loop that continues to check if SSID and PASSWORD are still empty or invalid, blinking an LED to indicate activity while waiting for valid SSID and password.
4. wifi.wiFiSetup(SSID, PASSWORD);:
   * Once a valid SSID and password are received, it calls the wiFiSetup function from the wifi object to connect to the Wi-Fi network.
5. else { wifi.wiFiSetup(SSID, PASSWORD); }:
   * If SSID and PASSWORD are already valid (i.e., not empty), it directly proceeds to set up the Wi-Fi connection.

**wiFi.mqttSetup(ServerMQTT,MqttPort) :-**

1. client.setServer(MqttSever, MqttPort);:
   * Configures the MQTT client to connect to the specified MQTT server (MqttSever) and port (MqttPort).
2. client.setCallback(MQTT\_Pull);:
   * Sets the callback function (MQTT\_Pull) that will be invoked whenever a new message is received from the MQTT server.



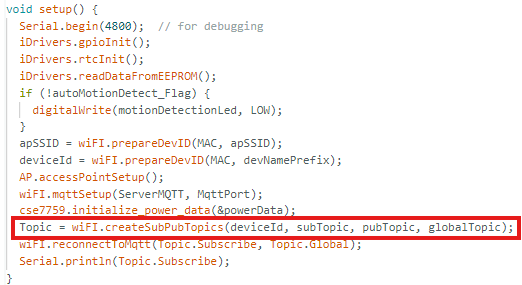
**cse7759.initialize\_power\_data(&powerData)** :-

1. memset(data, 0, sizeof(PowerData));:
   * Clears the memory of the data object (of type PowerData) by setting all its bytes to 0.
2. data->test\_flag = 0;:
   * Sets the test\_flag field of the data object to 0.
3. data->checksum = 255;:
   * Sets the checksum field of the data object to 255.
4. data->ifDataIsOk = false;:
   * Sets the ifDataIsOk field of the data object to false.



**Topic = wiFI.createSubPubTopics(deviceId, subTopic, pubTopic, globalTopic) :-**

1. subPubTopics Topics;:
   * Declares a variable Topics of type subPubTopics, which will hold the subscription and publication topic data.
2. Topics.Subscribe = devID + SubTopic;:
   * Concatenates the devID with the SubTopic string and stores the result in the Subscribe field of Topics.
3. Topics.Publish = devID + PubTopic;:
   * Concatenates the devID with the PubTopic string and stores the result in the Publish field of Topics.
4. Topics.Global = globalTopic + SubTopic;:
   * Concatenates the globalTopic with the SubTopic string and stores the result in the Global field of Topics.
5. return Topics;:
   * Returns the Topics object containing the formatted subscription, publication, and global topics.



**wiFI.reconnectToMqtt(Topic.Subscribe, Topic.Global):-**

1. if (!client.connected()) {:
   * Checks if the MQTT client is not currently connected to the server.
2. if (client.connect(prepareDevID(MAC, devNamePrefix).c\_str())) {:
   * Attempts to reconnect to the MQTT server using a device ID generated by the prepareDevID function, which uses the MAC address and device name prefix.
3. client.subscribe(subTopic.c\_str());:
   * Subscribes to the specified subTopic for receiving MQTT messages.
4. client.subscribe(globalTopic.c\_str());:
   * Subscribes to the specified globalTopic for receiving MQTT messages



**Main Loop for Wi-Fi, MQTT, Power Data, RTC, and Auto-Motion Detection**

**wiFI.clientLoop() :-**

* The clientLoop function simply calls client.loop(), which processes incoming and outgoing MQTT messages, maintaining the connection and handling communication with the MQTT broker.





**powerData = cse7759.validateTheCSEChipData(cse7759.read\_power\_data(), relayLoadStaus):-**

1. Initialize Variables: Calculates expected power and current based on voltage and power data.
2. Validate Data: Checks if power or current values are abnormally high, or if voltage is too low, to detect errors.
3. Handle Invalid Data: If data fails validation, it marks it as invalid and reverts to previous valid values.
4. Handle Valid Data: If data passes validation, it marks it as valid and updates previous values with current readings.
5. Return Data: Returns the powerData structure, now with a validation status and any adjustments.

This ensures only valid data is used for further processes.

**unsigned long currentTime = millis() :-**

* 1-Minute Interval Check: The code checks if 1 minute (60000 milliseconds) has passed since the last data publishing.
* Immediate Publishing on Specific Trigger: If responseOn200Request is true, it triggers an immediate publish, regardless of the 1-minute timer.
* Reset Timer After Publishing: After successful publishing, the timer (previousTime) resets, starting a new 1-minute countdown.



**wiFI.wiFiLinkCheck():-**

* Check Wi-Fi Status: It verifies whether the Wi-Fi connection is currently active by checking WiFi.status().
* Prevent Further Actions if Disconnected: If the device isn’t connected (WL\_CONNECTED is false), the function returns false, stopping further processes dependent on Wi-Fi, such as MQTT publishing.
* Reliability: It adds reliability by rechecking the connection status each time before attempting critical tasks, like connecting to MQTT or publishing data.
* Trigger Reconnection: If Wi-Fi is disconnected, this check helps initiate reconnection attempts, improving connection stability for ongoing operations.





**if (isApStarted) {**

**wiFiRetryCounter = 0;**

**isApStarted = false;**

**AP.stopApServer();**

**AP.stopApWiFi();**

**} :-**

* Reset Retry Counter: wiFiRetryCounter is set to 0, indicating that the Wi-Fi connection is now stable, so retry attempts can be reset.
* Disable AP Mode Flag: isApStarted is set to false to indicate that AP mode is no longer needed since the device is now connected to Wi-Fi.
* Stop Access Point Services:
* AP.stopApServer() stops the AP’s web server, which might have been used to allow users to configure Wi-Fi settings when the device was disconnected.
* AP.stopApWiFi() disables the AP mode entirely, freeing resources and focusing on the connected Wi-Fi mode.

**wiFI.CheckMQTTConnection():-**

* The CheckMQTTConnection function checks if the MQTT client is connected. It returns true if the client is connected, and false if it is not.





**if (responseOn200Request) {  wiFI.publishMqttMsg(Topic.Publish, "300", powerData.voltage, powerData.current, powerData.power, relayLoadStaus); responseOn200Request = false;**

**}** :- If responseOn200Request is true, publish a specific MQTT message and reset the flag.

The publishMqttMsg function builds a message string containing device ID, voltage, current, power, and load status.

Sends the formatted message over the specified MQTT topic.

**else {wiFI.publishMqttMsg(Topic.Publish, deviceId, powerData.voltage, powerData.current, powerData.power, relayLoadStaus); previousTime = currentTime; } :-**

* Publish Data: If responseOn200Request is not triggered, it publishes regular device data (voltage, current, power, and load status) to the specified MQTT topic.
* Reset Timer: Updates previousTime to the current time, restarting the 1-minute interval for the next scheduled publish.



**else { if (!wiFI.CheckMQTTConnection()){  wiFI.reconnectToMqtt(Topic.Subscribe, Topic.Global); } else {delay(100); } } :-**

if the MQTT client is connected:

* If not connected: It attempts to reconnect and subscribe to specified topics (subTopic and globalTopic).
* If connected: It simply waits briefly (100 ms).

CheckMQTTConnection() function confirms connection status, while reconnectToMqtt() handles reconnection and topic subscription if needed.

