Report on Libraries Used in ESP32 Project

1. WiFi.h

Purpose: The WiFi.h library is used to enable Wi-Fi connectivity on the ESP32 module. It allows the ESP32 to connect to a Wi-Fi network, act as an access point, and communicate over TCP/IP protocols.

Key Features:

- Connects to a Wi-Fi network as a client or access point.
- Provides functions to check connection status.
- Supports TCP, UDP, and HTTP communication.

Common Functions:

- WiFi.begin(ssid, password); - Connects to a Wi-Fi network.
- WiFi.status(); - Checks the connection status.
- WiFi.localIP(); - Retrieves the assigned IP address.

---

2. WebServer.h

Purpose: The WebServer.h library provides an easy way to run a web server on the ESP32. It allows handling HTTP requests and serving webpages directly from the ESP32.

Key Features:

- Handles GET and POST HTTP requests.
- Supports parameter handling in HTTP requests.
- Enables creating interactive web applications using ESP32.

Common Functions:

- server.on("/", handleRoot); - Defines a handler for the root URL.
- server.begin(); - Starts the web server.
- server.handleClient(); - Handles incoming client requests.

---

3. EmonLib

Purpose: EmonLib is the OpenEnergyMonitor library used for measuring voltage, current, and power in energy monitoring applications.

Key Features:

- Supports measuring AC voltage and current.
- Computes real power, apparent power, and power factor.
- Uses an ADC to sample electrical signals.

Common Functions:

- EnergyMonitor emon1; - Creates an instance of the energy monitor.
- emon1.current(pin, calibration); - Configures current measurement.
- emon1.calcIrms(samples); - Calculates the RMS current.

---

4. NTPClient.h

Purpose: The NTPClient.h library is used to synchronize time with an NTP (Network Time Protocol) server, ensuring accurate timekeeping on the ESP32.

Key Features:

- Fetches the current time from an NTP server.
- Supports different time zones and daylight savings time.
- Uses UDP communication for time synchronization.

Common Functions:

- WiFiUDP ntpUDP; - Initializes a UDP instance for NTP communication.
- NTPClient timeClient(ntpUDP, "pool.ntp.org"); - Creates an NTP client instance.
- timeClient.update(); - Updates the time from the NTP server.
- timeClient.getFormattedTime(); - Retrieves the current time as a formatted string.

---

5. ESP_Mail_Client.h

Purpose: ESP_Mail_Client.h is a library for sending and receiving emails using ESP32. It supports SMTP, IMAP, and POP3 protocols.

Key Features:

- Sends emails using SMTP.
- Supports attachments and HTML emails.
- Can fetch emails from IMAP and POP3 servers.

Common Functions:

- SMTPData smtpData; - Creates an instance for SMTP data.
- smtpData.setLogin(server, port, email, password); - Configures the email server and credentials.
- smtpData.setSender("ESP32", sender_email); - Sets the sender details.
- smtpData.setMessage("Message body", false); - Defines the email content.
- MailClient.sendMail(smtpData); - Sends the email.

Conclusion

These libraries provide essential functionality for ESP32-based projects, enabling Wi-Fi communication, web server hosting, energy monitoring, time synchronization, and email functionality. Their integration helps in developing robust IoT applications efficiently.

Report on Hardware Configuration

1. Voltage Measurement

- **Pin:** VOLTAGE_PIN (35)
- **Purpose:** Used to measure the AC voltage via a voltage sensor.
- **Calibration Factor:** vCalibration = 90.0
- **Description:** This pin is connected to the voltage sensor (e.g., ZMPT101B) and is used for analog voltage sampling.

2. Current Measurement

- **Pin:** CURRENT_PIN (34)
- **Purpose:** Used to measure the AC current via a current sensor.
- **Calibration Factor:** currCalibration = 6.0
- **Description:** This pin is connected to a current sensor (e.g., ACS712) to measure the current flowing through the circuit.

3. Buzzer

- **Pin:** BUZZER_PIN (5)
- **Purpose:** Used for audible alerts in case of threshold exceedance or system notifications.
- **Description:** The buzzer provides warning signals when predefined conditions are met.

4. Button

- **Pin:** BUTTON_PIN (4)
- **Purpose:** Used for user input, such as resetting the system or triggering an event.
- **Description:** This button can be programmed for various control functions.

5. LED Indicators

Red LED

- **Pin:** RED_LED (18)
- **Purpose:** Indicates slab chainged to PEAK.

Yellow LED

- **Pin:** YELLOW_LED (19)
- **Purpose:** ndicates slab chainged to OFF-PEAK

Green LED

- **Pin:** GREEN_LED (21)
- **Purpose:** ndicates slab chainged to NORMAL

Conclusion

The hardware configuration includes voltage and current sensors connected to analog pins for measurement, an alert system with a buzzer and LED indicators, and a user button for interaction. The calibration factors ensure accurate measurement of electrical parameters.

Report on Network and Email Configuration

1. Network Configuration

Wi-Fi Connection

- **SSID:** wifi_ssid
- **Password:** (Hidden for security reasons)
- **Purpose:** The ESP32 connects to the specified Wi-Fi network to enable internet access for cloud communication and remote monitoring.
- **Security Consideration:**
    - Avoid hardcoding Wi-Fi credentials in source code.
    - Store credentials securely using SPIFFS, EEPROM, or environment variables.

Access Point (AP) Mode

- **SSID:** ap_ssid (ESP32_EnergyMeter)
- **Password:** (Hidden for security reasons)
- **Purpose:** Enables ESP32 to act as an access point, allowing direct device connections without an external Wi-Fi network.
- **Use Case:** Useful for configuring the device in standalone mode or when no external Wi-Fi is available.
- **Security Consideration:** Use a strong password to prevent unauthorized access.

---

2. Email Configuration

SMTP Server Details

- **SMTP Server:** smtp.gmail.com
- **Port:** 465 (SSL secured)

Sender Details

- **Email:** sender_email
- **Password:** (Hidden for security reasons)
- **Purpose:** The ESP32 sends email notifications regarding energy usage, alerts, or reports.
- **Security Consideration:**
    - Use App Passwords instead of real email passwords.
    - Implement OAuth2 for better security.

Recipient Details

- **Recipient Email:** Recipient_email

- **Recipient Name:** Recipient_name
- **Purpose:** Receives energy consumption updates and alerts.

Conclusion

This configuration enables network connectivity and email notifications for the ESP32-based Smart Energy Meter. To enhance security, it is recommended to:

- Avoid storing sensitive credentials in plain text.
- Use secure authentication mechanisms for email.
- Implement encryption where necessary to protect data transmission.

Report on Global Objects and Energy Variables

1. Global Objects

Energy Monitor Object

- **Object:** EnergyMonitor emon;
- **Purpose:**
    - Used for energy measurement.
    - Part of the EmonLib library, which helps measure voltage, current, and power.

Web Server Object

- **Object:** WebServer server(80);
- **Purpose:**
    - Creates an HTTP server on port 80.
    - Handles requests for the energy meter's web interface.

NTP (Network Time Protocol) Objects

- **Object:** WiFiUDP ntpUDP;
- **Purpose:** Creates a UDP instance for network time synchronization.
- **Object:** NTPClient timeClient(ntpUDP, "pool.ntp.org", 19800, 60000);
- **Purpose:**
    - Synchronizes the ESP32's internal clock with an NTP server.
    - Uses pool.ntp.org as the time server.
    - 19800 refers to the time zone offset in seconds (UTC+5:30 for IST).
    - 60000 represents the update interval in milliseconds (60 seconds).

SMTP Session Object

- **Object:** SMTPSession smtp;
- **Purpose:**
    - Used for sending emails via the ESP_Mail_Client library.
    - Helps in sending alerts related to energy consumption.

---

2. Energy Variables

Measurement Variables

- **voltage** (float): Stores the measured voltage value.
- **current** (float): Stores the measured current value.

- **power** (float): Stores the calculated power consumption.
- **kWh** (float): Stores the accumulated energy consumption in kilowatt-hours.
- **cost** (float): Stores the calculated energy cost based on usage.

Slab and Alert Tracking

- **currentSlab** (String): Stores the current time-based tariff slab (e.g., Normal, Peak, Off-Peak).
- **lastSentSlab** (String): Stores the last slab for which an alert was sent to prevent duplicate notifications.
- **powerAlertSent** (bool): Prevents repeated power alerts.
- **dailyEnergyAlertSent** (bool): Tracks if the daily energy report has already been sent.
- **peakAlertSent** (bool): Ensures peak-time alerts are not sent multiple times.

Timing Variables

- **lastMillis** (unsigned long): Stores the last recorded time in milliseconds, used for timing operations.
- **lastDay** (int): Stores the last recorded day to track daily operations (e.g., resetting energy counters at midnight).

Conclusion

The global objects and variables play a crucial role in managing the ESP32-based Smart Energy Meter, ensuring real-time monitoring, web interfacing, time synchronization, and automated alerting mechanisms. Proper use of these variables helps in optimizing system efficiency and avoiding redundant alerts.

Report on Web Dashboard HTML

1. Overview

The provided HTML code represents a web-based dashboard for the **Smart Energy Meter**. It dynamically displays real-time energy consumption parameters, including voltage, current, power, energy usage, cost, and the current time-based slab.

2. Key Features

- **Responsive Design:** Uses CSS Grid (grid-template-columns: repeat(auto-fit, minmax(280px, 1fr))) to adjust layout based on screen size.
- **Real-time Updates:** Fetches data from the /data endpoint every 2 seconds (setInterval(updateData, 2000);).
- **Dynamic Slab Highlighting:** Changes the background color of the slab box based on the current tariff slab (Normal, Peak, Off-Peak).
- **Smooth UI Enhancements:** Includes hover effects and a blurred background for a modern aesthetic.

3. Structure Breakdown

a) HTML Elements

- **Header (<h1>)**: Displays the dashboard title.
- **Container ( )**: Houses multiple box elements for energy parameters.
- **Footer (<footer>)**: Displays a copyright notice.

b) CSS Styling

- **Background:** Uses a gradient from #1e1e2e to #3a3a55.
- **Grid Layout:** Ensures flexible arrangement of energy metrics.
- **Box Styling:**
    - border-radius: 15px for rounded corners.
    - box-shadow: 0 6px 12px rgba(0, 0, 0, 0.3); for depth.
    - backdrop-filter: blur(10px); for a frosted-glass effect.
    - Dynamic colors for slabs:
        - **Normal:** rgba(0, 200, 83, 0.8) (Green)
        - **Peak:** rgba(244, 67, 54, 0.8) (Red)
        - **Off-Peak:** rgba(255, 152, 0, 0.8) (Orange)

c) JavaScript Functionality

- **AJAX Fetching (fetch("/data"))**:
    - Retrieves JSON data from the ESP32 server.

    - Updates UI elements dynamically (innerText).

- **Dynamic Class Assignment:**
    - slabBox.className = "box " + (data.slab === "Normal" ? "normal" : data.slab === "Peak" ? "peak" : "offpeak");
    - Updates the background color based on the current slab.
- **Automatic Updates:**
    - Uses setInterval(updateData, 2000); to refresh data every 2 seconds.

4. Security Considerations

- **CORS & Access Control:** Ensure that the /data endpoint is secured against unauthorized access.
- **Data Validation:** The ESP32 backend should sanitize and validate data before sending it to the frontend.

5. Conclusion

This dashboard provides an intuitive and visually appealing interface for monitoring energy usage in real time. The combination of responsive design, smooth UI elements, and efficient data fetching makes it a robust solution for smart energy monitoring.

Report on Buzzer and LED Control Functions

1. Overview

The given functions control the **buzzer** and **LED indicators** in the Smart Energy Meter system. The buzzer serves as an alert mechanism, while the LEDs visually indicate the current energy tariff slab.

2. Function Breakdown

a) triggerBuzzer()

- **Purpose:** Activates the buzzer for 2 seconds as an alert mechanism.
- **Process:**
    1. Sets BUZZER_PIN **HIGH** to turn the buzzer on.
    2. Delays execution for **2000 milliseconds (2 seconds)**.
    3. Sets BUZZER_PIN **LOW** to turn the buzzer off.
- **Use Cases:**
    - Alerts for **overconsumption, high power usage, or warnings**.
    - Can be triggered when exceeding predefined thresholds.

b) updateLEDs()

- **Purpose:** Updates LED indicators based on the current time-based slab.
- **Process:**
    - Uses **digitalWrite()** to control LED states:
        - **RED_LED** → ON if currentSlab == "Peak" (High tariff period)
        - **YELLOW_LED** → ON if currentSlab == "Off-Peak" (Low tariff period)
        - **GREEN_LED** → ON if currentSlab == "Normal" (Standard tariff period)
- **Use Cases:**
    - Provides **visual feedback** for the current energy cost slab.
    - Helps users monitor consumption at a glance.

3. Potential Improvements

- **Non-Blocking Delay:** Instead of delay(2000), use millis() for better execution flow.
- **Buzzer Modulation:** Implement different tones for various alerts.
- **LED Blinking:** Blink LEDs during specific conditions (e.g., overuse warnings).

4. Conclusion

These functions efficiently handle **alerting and visual indications** in the Smart Energy Meter. Optimizing them with non-blocking techniques can improve system responsiveness and functionality.

Report on Email Alert Function

1. Overview

The sendEmailAlert function is responsible for sending **email notifications** when specific energy-related alerts are triggered. It uses the ESP_Mail_Client library to connect to an SMTP server and send emails in **HTML format**.

2. Function Breakdown

a) Email Session Setup

- **Object:** ESP_Mail_Session session;
- **Configuration:**
    - Sets SMTP server details (SMTP_server and SMTP_Port).
    - Uses sender_email and sender_password for authentication.
    - session.login.user_domain = ""; (default, not used here).

b) Message Composition

- **Object:** SMTP_Message message;
- **Sender:**
    - Name: "ESP32 Smart Meter"
    - Email: sender_email
- **Subject Line:**
    - Dynamically appends alertType (e.g., "Energy Alert: Peak Consumption").
- **Recipient:**
    - Name: Recipient_name
    - Email: Recipient_email

c) HTML Email Content

- **Dynamic Color Coding:**
    - Peak → **Red**
    - Off-Peak → **Orange**
    - Normal → **Green**
- **Email Body Includes:**
    - Time (from NTPClient).
    - Voltage, Current, Power, Energy, Cost.
    - **Current Tariff Slab** (colored based on type).

d) Sending the Email

1. **SMTP Connection:** smtp.connect(&session);
2. **Sending the Email:** MailClient.sendMail(&smtp, &message);
3. **Error Handling:**
    - If email fails, prints error reason: Serial.println("Error sending Email: " + smtp.errorReason());

---

3. Potential Improvements

- **Security:**
    - Use **App Passwords** instead of storing plain text email passwords.
    - Consider OAuth authentication for improved security.
- **Error Handling Enhancements:**
    - Implement a **retry mechanism** if sending fails.
    - Store failure logs for debugging.
- **Customization:**
    - Allow users to define custom thresholds for email alerts.
    - Add more formatting styles to enhance email readability.

---

4. Conclusion

This function effectively notifies users about energy consumption through **real-time email alerts**. With additional security improvements and enhanced logging, it can be more robust and reliable for IoT-based smart metering applications.

Report on Sensor Reading Function

1. Overview

The readSensors() function is responsible for:

- Measuring **voltage, current, and power** using the EmonLib energy monitoring library.
- **Calculating energy consumption (kWh)** over time.
- **Tracking time** using an NTP client to determine the current hour and day.

---

2. Function Breakdown

a) Voltage, Current, and Power Measurement

```
emon.calcVI(20, 2000);
voltage = emon.Vrms;
current = emon.Irms;
power = voltage * current;
```

- **emon.calcVI(20, 2000);**
    - Performs **voltage and current calculations** over **20 cycles** with a **2000ms timeout**.
    - Outputs:
        - emon.Vrms → **Voltage (RMS value)**
        - emon.Irms → **Current (RMS value)**
- **Power Calculation:** power = voltage * current;
    - Uses **Ohm's Law** to compute real-time power consumption.

---

b) Energy Consumption (kWh) Calculation

```
unsigned long currentMillis = millis();
float elapsedHours = (currentMillis - lastMillis) / 3600000.0;
kWh += (power * elapsedHours) / 1000.0;
lastMillis = currentMillis;
```

- **Time Tracking:**
    - Uses millis() to determine **elapsed time** since the last reading.
    - Converts elapsed time from **milliseconds to hours** (/ 3600000.0).
- **Energy Calculation:**
    - Energy (kWh) = **Power (W) × Time (hours) / 1000**.

- Accumulates kWh readings over time for total energy usage.

---

c) Time Synchronization & Slab Tracking

timeClient.update();

int hour = timeClient.getHours();

int currentDay = timeClient.getDay();

- **NTP Client Update (timeClient.update();)**
  - Synchronizes the ESP32 with **network time**.
- **Extracts:**
  - hour → Used to determine the **current energy slab**.
  - currentDay → Can be used for **daily energy tracking**.

---

3. Potential Improvements

- **Optimized Sampling:**
  - Adjust calcVI() parameters based on **AC cycle stability**.
- **Interrupt-Based Timer:**
  - Instead of millis(), use a **Timer Interrupt** for energy calculation to avoid drift over time.
- **Data Logging:**
  - Store **energy consumption per hour/day** for historical analysis.

---

4. Conclusion

The readSensors() function effectively monitors real-time **voltage, current, power, and energy consumption**, ensuring accurate tracking of power usage. By implementing **timer-based updates and logging mechanisms**, its accuracy and efficiency can be further improved for smart energy management applications.

Report on Energy Slab Calculation

1. Overview

The **slab calculation** determines the cost of electricity consumption based on the time of day. It categorizes power usage into three slabs—**Normal, Peak, and Off-Peak**—with different pricing structures.

---

2. Slab Calculation Logic

```
if (hour >= 6 && hour < 18) {
   cost = kWh * 5.00;
   currentSlab = "Normal";
} else if (hour >= 18 && hour < 22) {
   cost = kWh * 7.00;
   currentSlab = "Peak";
} else {
   cost = kWh * 3.50;
   currentSlab = "Off-Peak";
}
```

a) Normal Slab

- **Time Range:** 6:00 AM - 6:00 PM
- **Rate:** ₹5.00 per kWh
- **Usage Scenario:**
    - General daytime consumption (work hours, household activities).

b) Peak Slab

- **Time Range:** 6:00 PM - 10:00 PM
- **Rate:** ₹7.00 per kWh (highest tariff)
- **Usage Scenario:**
    - Evening peak demand (lighting, cooking, entertainment).

c) Off-Peak Slab

- **Time Range:** 10:00 PM - 6:00 AM
- **Rate:** ₹3.50 per kWh (lowest tariff)
- **Usage Scenario:**
    - Nighttime consumption (charging devices, lower overall demand).

---

3. Potential Improvements

- **Dynamic Pricing:**
    - Allow users to configure rates based on utility company tariffs.
- **Seasonal Adjustments:**
    - Modify slab times for summer/winter power demands.
- **Data Logging & Analysis:**
    - Store slab-wise consumption data for cost optimization.
- **Visual Indicators:**
    - Implement **LED indicators or web UI alerts** to notify users of current slab changes.

---

4. Conclusion

The slab-based tariff system ensures **cost-effective power usage** by charging higher rates during peak hours and encouraging off-peak consumption. Enhancing it with **dynamic adjustments and logging mechanisms** can make it even more efficient for smart energy monitoring applications.

Report on Energy Slab Calculation and Alerts

1. Overview

The **slab calculation** determines the cost of electricity consumption based on the time of day. It categorizes power usage into three slabs—**Normal, Peak, and Off-Peak**—with different pricing structures. Additionally, an alert system is implemented to notify users of high power usage, daily energy limits, and slab changes.

---

2. Slab Calculation Logic

```
if (hour >= 6 && hour < 18) {
   cost = kWh * 5.00;
   currentSlab = "Normal";
} else if (hour >= 18 && hour < 22) {
   cost = kWh * 7.00;
   currentSlab = "Peak";
} else {
   cost = kWh * 3.50;
   currentSlab = "Off-Peak";
}
```

a) Normal Slab

- **Time Range:** 6:00 AM - 6:00 PM
- **Rate:** ₹5.00 per kWh
- **Usage Scenario:**
    - General daytime consumption (work hours, household activities).

b) Peak Slab

- **Time Range:** 6:00 PM - 10:00 PM
- **Rate:** ₹7.00 per kWh (highest tariff)
- **Usage Scenario:**
    - Evening peak demand (lighting, cooking, entertainment).

c) Off-Peak Slab

- **Time Range:** 10:00 PM - 6:00 AM
- **Rate:** ₹3.50 per kWh (lowest tariff)
- **Usage Scenario:**

- Nighttime consumption (charging devices, lower overall demand).

---

3. Alert System Implementation

```
// Alerts
if (power > 1000 && !powerAlertSent) {
   triggerBuzzer();
   sendEmailAlert("High Power Usage (Above 1000W)");
   powerAlertSent = true;
} else if (power <= 1000) powerAlertSent = false;

if (currentDay != lastDay) {
   kWh = 0;
   dailyEnergyAlertSent = false;
   lastDay = currentDay;
}
if (kWh > 7 && !dailyEnergyAlertSent) {
   triggerBuzzer();
   sendEmailAlert("Daily Energy Limit Exceeded");
   dailyEnergyAlertSent = true;
}

if (currentSlab != lastSentSlab) {
   sendEmailAlert("Slab Changed to " + currentSlab);
   lastSentSlab = currentSlab;
   if (currentSlab == "Peak") {
      triggerBuzzer();
      peakAlertSent = true;
   }
}

updateLEDs();
```

a) High Power Usage Alert

- **Condition:** Power exceeds **1000W**.
- **Action:**
    - Triggers **buzzer**.
    - Sends **email alert**.
    - Resets when power goes below threshold.

b) Daily Energy Limit Alert

- **Condition:** Daily consumption exceeds **7 kWh**.
- **Action:**
    - Triggers **buzzer**.
    - Sends **email alert**.
    - Resets daily at midnight.

c) Slab Change Alert

- **Condition:** Change in time-based slab.
- **Action:**
    - Sends **email alert** with new slab.
    - If entering **Peak slab**, triggers **buzzer**.

---

4. Potential Improvements

- **Dynamic Pricing:**
    - Allow users to configure rates based on utility company tariffs.
- **Seasonal Adjustments:**
    - Modify slab times for summer/winter power demands.
- **Data Logging & Analysis:**
    - Store slab-wise consumption data for cost optimization.
- **Enhanced Alert Customization:**
    - Enable users to set personalized power and energy thresholds.

---

5. Conclusion

The slab-based tariff system ensures **cost-effective power usage** by charging higher rates during peak hours and encouraging off-peak consumption. The alert mechanism enhances this by **notifying users of critical events**, helping them **optimize energy consumption** and **prevent excessive power usage**.

Future improvements can focus on **customizable alerts, data analytics, and adaptive tariff structures**.

Data Handling Function in Smart Energy Consumption Meter

1. Introduction

The handleData() function is a key component of the **Smart Energy Consumption Meter using IoT**, responsible for collecting real-time sensor readings and formatting them into a JSON response for transmission. This function ensures that power consumption data is available for external systems, such as the web dashboard or MQTT broker, enabling remote monitoring and analysis.

2. Function Overview

The handleData() function performs the following tasks:

1. **Reading Sensor Data**: Calls readSensors(), which retrieves voltage, current, power, and energy readings from the ZMPT101B and ACS712 sensors.
2. **Fetching the Current Time**: Formats the real-time clock (RTC) data into a human-readable HH:MM:SS format.
3. **Constructing a JSON Response**: Combines the sensor readings, time, and cost data into a structured JSON format.
4. **Sending the Data**: Returns the JSON response to an HTTP request, making it available for real-time monitoring.

3. Code Breakdown

```
void handleData() {
  readSensors();
  char timeStr[10];
  snprintf(timeStr, sizeof(timeStr), "%02d:%02d:%02d",
      timeClient.getHours(), timeClient.getMinutes(), timeClient.getSeconds());

  String jsonData = "{";
  jsonData += "\"voltage\":" + String(voltage, 2) + ",";
  jsonData += "\"current\":" + String(current, 2) + ",";
  jsonData += "\"power\":" + String(power, 2) + ",";
  jsonData += "\"energy\":" + String(kWh, 3) + ",";
  jsonData += "\"cost\":" + String(cost, 2) + ",";
  jsonData += "\"slab\":\"" + currentSlab + "\",";
  jsonData += "\"time\":\"" + String(timeStr) + "\"}";

  server.send(200, "application/json", jsonData);
```

}

4. JSON Data Format

The generated JSON follows this structure:


```
{
  "voltage": 230.50,
  "current": 5.20,
  "power": 1198.60,
  "energy": 3.425,
  "cost": 1.50,
  "slab": "Normal",
  "time": "14:30:15"
}
```


This structure enables easy parsing by client applications such as a web dashboard or mobile app.

5. Function Significance

- **Real-time Monitoring**: Provides up-to-date energy consumption statistics.
- **Remote Access**: Enables data retrieval over the web.
- **Billing and Analysis**: Helps users track electricity costs based on different time-based slabs.

6. Conclusion

The handleData() function plays a crucial role in enabling IoT-based energy monitoring. By formatting sensor data into JSON and serving it via an HTTP request, this function supports seamless integration with web dashboards and other analytics platforms.

Setup() Function Report: IoT-Based Smart Energy Meter

1. Introduction

The **setup()** function plays a critical role in initializing and configuring essential components of the IoT-based smart energy meter. These include WiFi connectivity, sensor calibration, time synchronization, and web server setup. This ensures seamless operation, real-time data collection, and efficient communication.

2. Hardware Configuration

Several hardware components are initialized within the **setup()** function to ensure proper system functionality:

- **BUZZER_PIN**: Configured as an output to provide auditory alerts for system notifications.
- **BUTTON_PIN**: Configured as an input with an internal pull-up resistor for user interaction.
- **RED_LED, YELLOW_LED, GREEN_LED**: Configured as outputs to indicate different system statuses.

3. Network Setup

The system establishes network connectivity by enabling both Access Point (AP) and Station (STA) modes:

- WiFi.mode(WIFI_AP_STA): Activates dual-mode operation for local and remote access.
- WiFi.softAP(ap_ssid, ap_password): Sets up a local access point for direct device communication.
- WiFi.begin(wifi_ssid, wifi_password): Connects to the specified WiFi network for internet access.
- A loop checks the connection status, ensuring a stable connection before proceeding.

4. Sensor Initialization

To accurately measure electrical parameters, the function initializes voltage and current sensors using the **emon** library:

- emon.voltage(VOLTAGE_PIN, vCalibration, 1.7): Configures the voltage sensor with a specified calibration factor.
- emon.current(CURRENT_PIN, currCalibration): Configures the current sensor with the appropriate calibration factor.

5. Time Synchronization

For accurate timestamping and scheduling, the function initializes an **NTP-based time client**:

- timeClient.begin(): Begins synchronization with an NTP (Network Time Protocol) server.
- A loop ensures that the time is successfully updated before proceeding.

6. Web Server Initialization

To facilitate remote monitoring and data access, the function sets up a local web server:

- server.on("/", []() { server.send(200, "text/html", webpage); }); - Serves the main webpage to users.
- server.on("/data", handleData); - Manages real-time data requests from users or other connected devices.
- server.begin(); - Starts the web server to handle incoming HTTP requests.

7. Conclusion

The **setup()** function is the backbone of the IoT-based smart energy meter, ensuring proper hardware initialization, reliable network connectivity, precise sensor calibration, synchronized timekeeping, and seamless web-based interaction. These foundational processes collectively enable real-time data monitoring and effective energy consumption tracking.

Report on Arduino Loop Function

1. Introduction The provided loop function is part of an embedded system designed for monitoring and reporting energy consumption. It integrates sensor reading, server communication, and a manual report trigger via a button press.

2. Functionality Overview The function executes continuously, performing the following tasks in each iteration:

- Handling client requests via the server.handleClient() function.
- Reading sensor data through the readSensors() function.
- Monitoring a button press to manually trigger an energy report.
- Sending an alert and activating a buzzer upon detecting a button press.

3. Breakdown of Function Execution

a. Handling Client Requests

server.handleClient();

This function processes incoming client requests, ensuring that the system remains responsive to external queries and commands.

b. Reading Sensor Data

readSensors();

This function gathers real-time data from connected sensors, such as voltage and current measurements, for further processing and reporting.

c. Detecting a Button Press

```
if (digitalRead(BUTTON_PIN) == LOW) {
  delay(50); // Simple debounce
```

This section monitors the state of a button connected to BUTTON_PIN. If the button is pressed (LOW state), the system introduces a small delay (50 ms) for debounce to avoid false triggers.

d. Confirming Button Press and Executing Actions

```
if (digitalRead(BUTTON_PIN) == LOW) {
  Serial.println("Manual Report Triggered");
  sendEmailAlert("Manual Energy Report");
  triggerBuzzer();
```

After the debounce delay, the button state is checked again to confirm the press. If confirmed:

- A message is printed to the serial monitor.
- The sendEmailAlert() function is called to send a manual energy report notification.
- The triggerBuzzer() function is activated, likely providing an audible confirmation.

e. Waiting for Button Release

while(!digitalRead(BUTTON_PIN)); // Wait for release

This loop ensures that the system waits until the button is released before proceeding, preventing repeated triggers from a single press.

4. Delay for Stability

delay(100);

A short delay (100 ms) is introduced at the end of each loop iteration to reduce excessive polling and stabilize execution.

5. Conclusion The function efficiently integrates server handling, sensor reading, and a manual energy report trigger mechanism. The use of debouncing and wait-for-release techniques ensures reliable operation, reducing false triggers. Future improvements could include implementing interrupt-based button handling for better responsiveness.