Assignment 03 - Smart Temperature Monitoring

January 2025

Contents

1 Temperature Monitoring Subsystem (ESP32)

The ESP32 microcontroller acts as the core of the temperature monitoring subsystem. It is responsible for temperature data acquisition, network management connectivity, and communication with external systems via MQTT. The system architecture is designed to ensure reliable data acquisition and efficient communication with the backend. This is achieved by a thread-safe task architecture that runs on a shared object that simultaneously serves as a data collection and dissemination framework between system tasks.

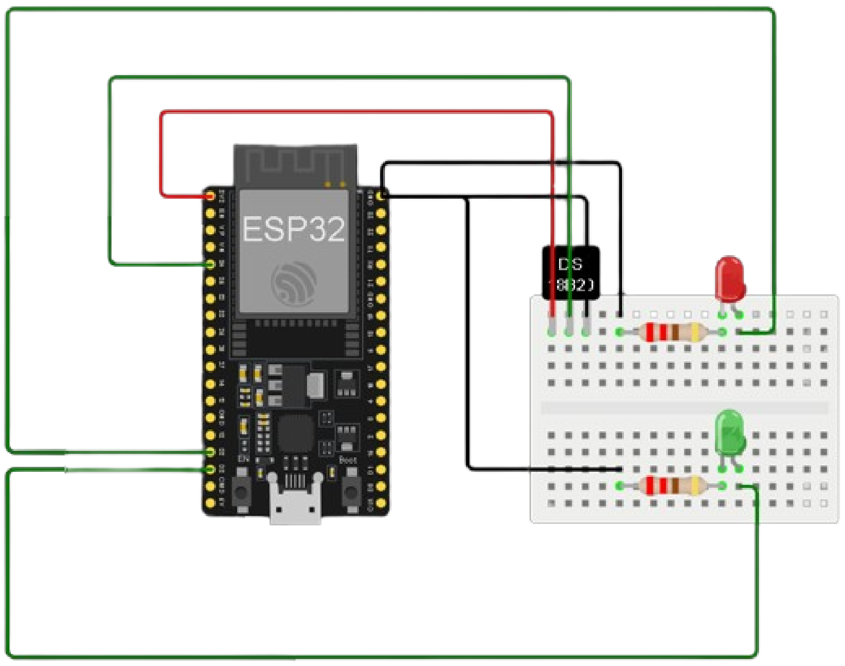


Figure 1: ESP32 Hardware Diagram

1.1 System Architecture

The ESP32-based subsystem consists of the following key components:

- **MqttTask**: MqttTask: Manages WiFi and MQTT connections, ensuring that the device remains connected to the network and is able to send data to the MQTT broker. It handles reconnections in the event of network failures and publishes temperature readings at a dynamic rate determined by the backend.

- **LedTask**: Controls LED indicators to provide visual feedback on the ESP32's connection status, indicating whether the system is properly connected to both WiFi and MQTT or if one of these connections is lost.
- **TemperatureTask**:Periodically reads temperature data from a sensor and updates the shared object (SharedState). The task runs on a timer, ensuring that readings are taken at the appropriate intervals and transmitted to the backend, all while being thread-safe to prevent races and inappropriate use of shared variables by the task.

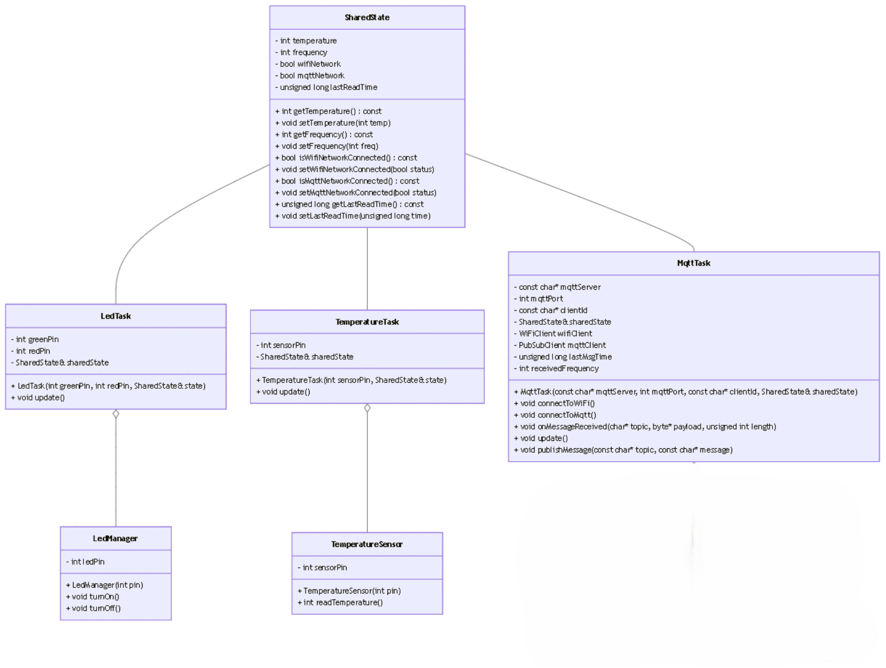

SharedState
- int temperature
- int frequency
- bool wifiNetwork
- bool mqttNetwork
- unsigned long lastReadTime
+ int getTemperature() : const
+ void setTemperature(int temp)
+ int getFrequency() : const
+ void setFrequency(int freq)
+ bool isWifiNetworkConnected() : const
+ void setWifiNetworkConnected(bool status)
+ bool isMqttNetworkConnected() : const
+ void setMqttNetworkConnected(bool status)
+ unsigned long getLastReadTime() : const
+ void setLastReadTime(unsigned long time)
LedTask
- int greenPin
- int redPin
- SharedState& sharedState
+ LedTask(int greenPin, int redPin, SharedState& state)
+ void update()
TemperatureTask
- int sensorPin
- SharedState& sharedState
+ TemperatureTask(int sensorPin, SharedState& state)
+ void update()
MqttTask
- const char* mqttServer
- int mqttPort
- const char* clientId
- SharedState& sharedState
- WiFiClient wifiClient
- PubSubClient mqttClient
- unsigned long lastMsgTime
- int receivedFrequency
+ MqttTask(const char* mqttServer, int mqttPort, const char* clientId, SharedState& sharedState)
+ void connectToWiFi()
+ void connectToMqtt()
+ void onMessageReceived(char* topic, byte* payload, unsigned int length)
+ void update()
+ void publishMessage(const char* topic, const char* message)
LedManager
- int ledPin
+ LedManager(int pin)
+ void turnOn()
+ void turnOff()
TemperatureSensor
- int sensorPin
+ TemperatureSensor(int pin)
+ int readTemperature()

Figure 2: Temperature Monitoring Subsystem Class Diagram

1.2 State Tasks Management

The ESP32 subsystem operates within a state-driven architecture, where each task transitions between states based on system conditions. The overall system is encapsulated within the **Active** state, which contains the three tasks:

- **MqttTask** switches between the WIFI CONNECT, MQTT CONNECT and CONNECTED states to ensure continuous network availability. The task is responsible, first of all, for establishing a connection with the selected WiFi network and, secondly, for connecting to the appropriate MQTT broker. Once the connection state is reached, it constantly monitors its state and handles incoming and outgoing messages.

- **LedTask** switches between NOT CONNECT and CONNECT, updating the LED indicators according to the state of the network.
- **TemperatureTask** switches between the NOT CONNECT and CONNECT states to manage temperature acquisition only when necessary, thus avoiding unnecessary measurements and access to shared resources.

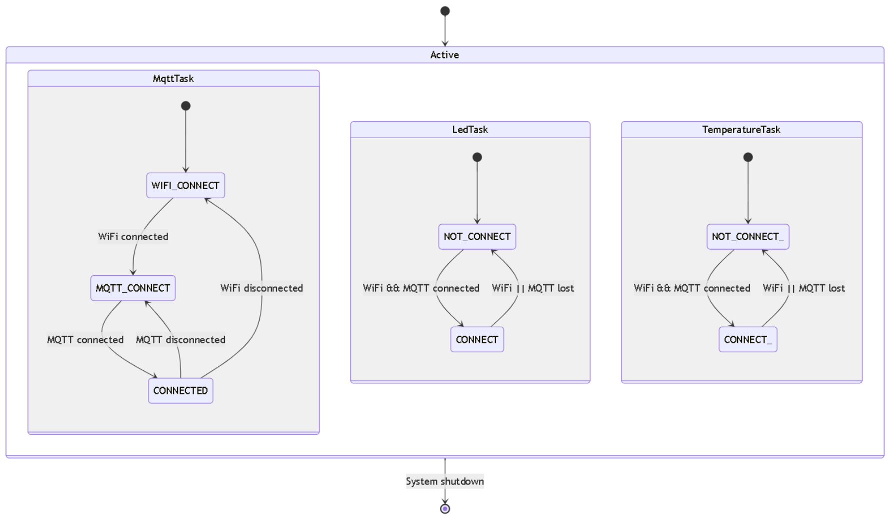


Figure 3: Temperature Monitoring Dubsystem Class Diagram

1.3 Data Communication

Once the temperature data is acquired, it is published in an MQTT topic through MqttTask. The update frequency is dynamically adjusted on the basis of commands received from the backend. The ESP32 ensures secure and reliable communication by implementing reconnection and error reporting mechanisms.

2 Control Unit subsystem (backend - running on a PC)

2.1 General system description

The diagram represents a temperature management system divided into various packages:

- **model**: Contains the main classes, such as `TempManager`, responsible for managing temperature states (`TempState`) and frequency, along with the `Logic` class, which coordinates operational modes (`ModeType`).

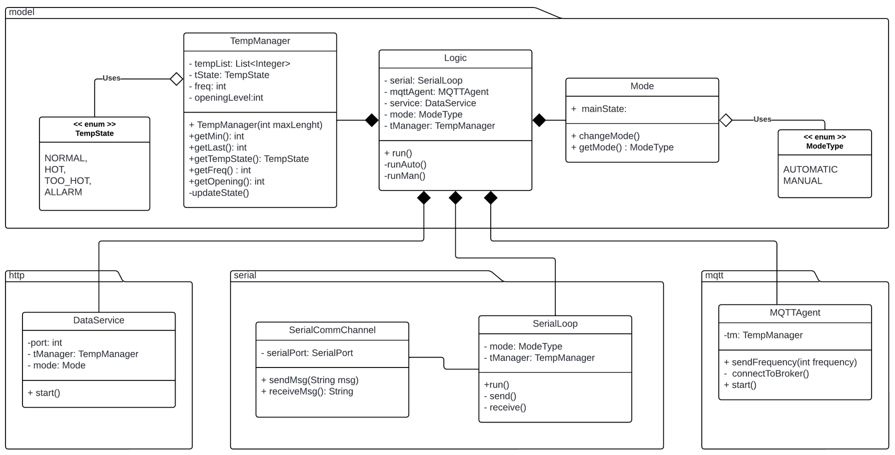


Figure 4: Class diagram

- **http**: Includes `DataService`, which connects the temperature manager (`TempManager`) and the current mode (`Mode`) via a network port.
- **serial**: Comprises the `SerialCommChannel` and `SerialLoop` classes, dedicated to communication and data transfer via the serial port.
- **mqtt**: Contains the `MQTTAgent` class, which manages the connection to the MQTT broker and the transmission of the update frequency.

2.2 Logic

The `Logic` class is the container for all connections and shared objects. It extends the `Thread` class and manages the core logic of the system. It is responsible for starting the HTTP, MQTT, and serial communication services. Based on the selected mode (Automatic or Manual), it performs various temperature management operations and updates the frequency based on the temperature state. `Logic` interacts with the `TempManager` and `Mode` classes, which encapsulate the system's states and values. These classes are used to enable communication between the logic and the classes dedicated to external communication. Since only one instance of each class is used, data consistency is guaranteed across all components of the project. The classes `DataService`, `MQTTAgent`, and `SerialLoop` are used to coordinate system activities and, based on the data they receive, modify the values stored in `TempManager` and `Mode`.

2.3 TempManager

As mentioned earlier, the `TempManager` class manages the temperature, storing historical values and determining the current temperature state. It uses the

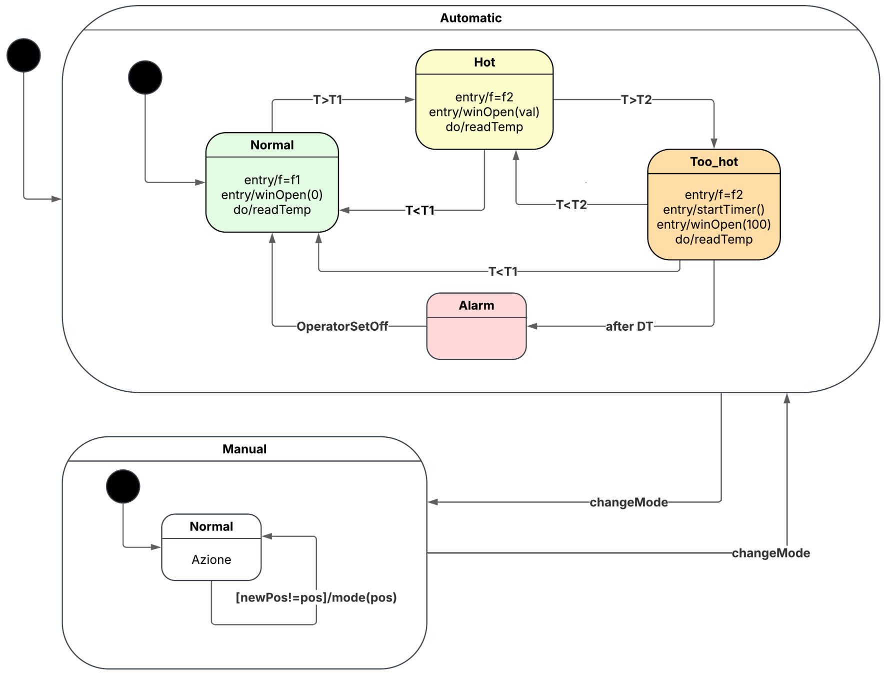


Figure 5: State diagram of the lgic class

constants defined in `Constants` to evaluate whether the temperature is normal, high, or too high. Additionally, it manages window opening and alarm activation. This class is essential for tracking the thermal state of the system and providing updated data to other components.

2.4 Mode

The `Mode` class manages the system's operating mode (Automatic or Manual). It provides methods to retrieve the current state and to change the mode. `Mode` is used by `Logic` to determine the system's behavior based on the selected mode. The mode can be changed by both the front-end interface and Arduino, but only one method is implemented to change the mode to ensure greater data accuracy and consistency.

2.5 DataService

The `DataService` class is an HTTP service that allows receiving and sending data related to temperature and the operating mode. It uses `TempManager` to retrieve temperature data and `Mode` to manage the mode. `DataService` exposes REST APIs for system interaction, enabling a front-end interface or other services to access and modify the system's state.

2.6 MQTTAgent

The `MQTTAgent` class manages communication with an MQTT broker to receive temperature data and send the update frequency. It uses `TempManager` to update the temperature data received from the broker. `MQTTAgent` is used by `Logic` to determine the frequency to send based on the current mode. This class is essential for integrating the system with external devices that communicate via the MQTT protocol.

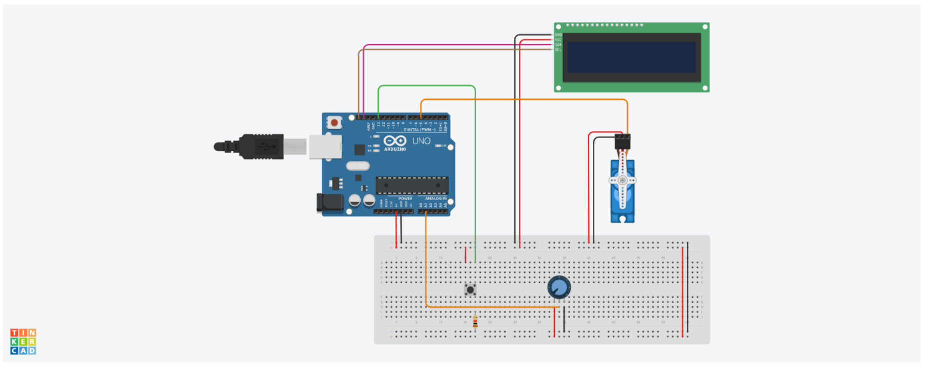
2.7 SerialLoop

The `SerialLoop` class extends `Thread` to operate independently of the main logic. It manages serial communication with an external device, such as an Arduino, to send and receive commands. It uses `TempManager` and `Mode` to synchronize the system's state with Arduino. `SerialLoop` is used by `Logic` to send commands to Arduino and receive feedback. This class is crucial for ensuring smooth and real-time communication with connected hardware devices.

2.8 Conclusion

The temperature management system is designed with a modular and well-structured architecture. The main classes interact to ensure efficient and flexible operation. The separation of responsibilities between the classes facilitates maintenance and extensibility. The system can operate in Automatic and Manual modes, manage alarms, and communicate with external devices using protocols such as MQTT, HTTP, and serial communication.

3 Window Controller subsystem (Arduino)



3.1 ModeTask

The `ModeTask` handles the toggling between manual and automatic modes using a physical button. The task works as follows:

- The task monitors the state of a physical button connected to the Arduino. When the button is pressed, the task toggles the system mode between `MANUAL` and `AUTOMATIC`.
- To prevent rapid mode switching due to button bouncing, the task uses a debouncing mechanism with a timer. The mode change is only registered if the button is pressed and the timer has elapsed.

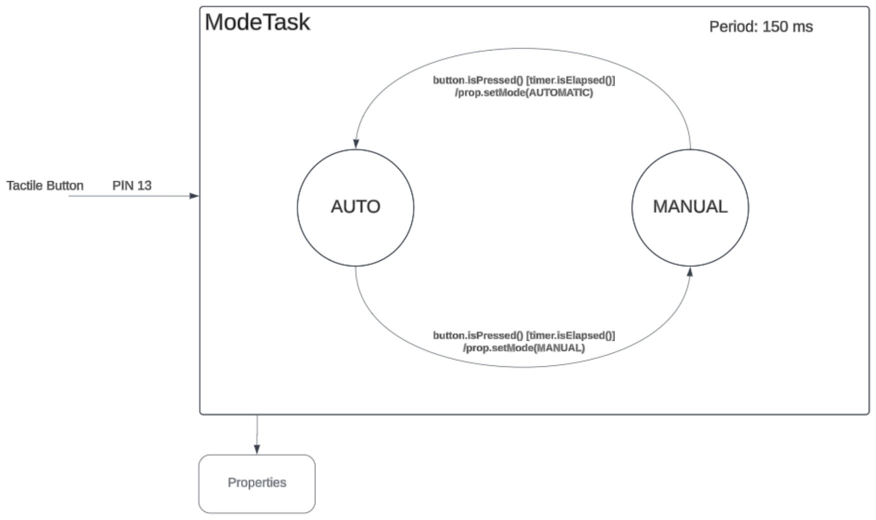

ModeTask
Period: 150 ms
button.isPressed() [timer.isElapsed()] /prop.setMode(AUTOMATIC)
Tactile Button
PIN 13
AUTO
MANUAL
button.isPressed() [timer.isElapsed()] /prop.setMode(MANUAL)
Properties

Figure 6: State diagram of ModeTask

3.2 GateTask

The `GateTask` is responsible for controlling the gate's servo motor in manual mode. It uses a potentiometer to determine the desired gate position and adjusts the servo accordingly. The task works as follows:

- The task continuously monitors the potentiometer's position in manual mode. If the potentiometer's position differs from the current gate position by more than 3 units, the task moves the gate to the new position and updates the shared `Properties` object with the new position.
- The task ensures smooth operation by only moving the gate when there is a significant change in the potentiometer's position, preventing unnecessary servo movements.

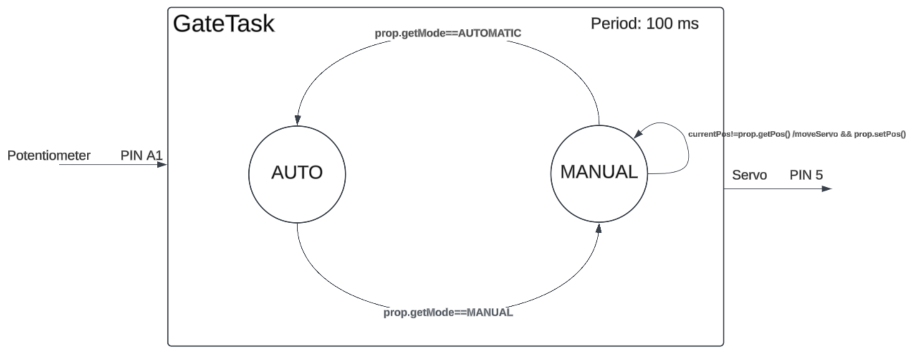


Figure 7: State diagram of GateTask

3.3 LCDTask

The `LCDTask` is responsible for displaying the current system mode, gate position, and temperature on an LCD screen. The task works as follows:

- The task continuously monitors the system mode, gate position, and temperature. If any of these values change, the task updates the LCD display to reflect the new state.
- In manual mode, the LCD displays both the gate position and the temperature. In automatic mode, only the gate position is displayed.

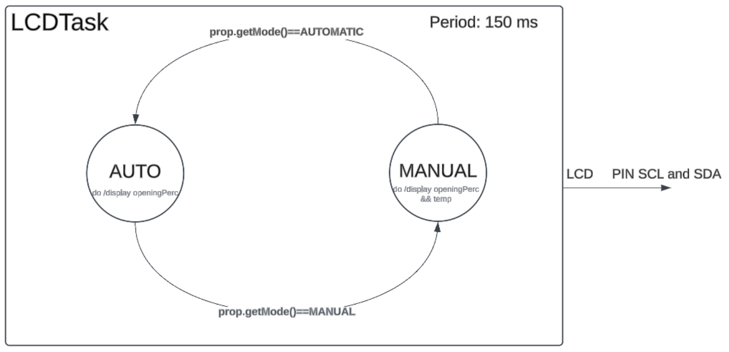


Figure 8: State diagram of LCDTask

3.4 SerialCommTask

The `SerialCommTask` manages serial communication for sending and receiving commands. The task works as follows:

- The task alternates between sending and receiving states. In the sending state, it transmits the current gate position or mode information over the serial connection.
- In the receiving state, the task listens for incoming messages. If a message is received, it parses the message and updates the system mode, position, or temperature accordingly.
- The task handles two types of incoming messages: `M` messages for manual mode and temperature, and `A` messages for automatic mode and position. Outgoing messages are formatted as `MXXX` for manual mode (where XXX is the opening level of the gate) or `A` for automatic mode.
- When the task receives a Modality that is different from the current one, it provides to update the object Properties.

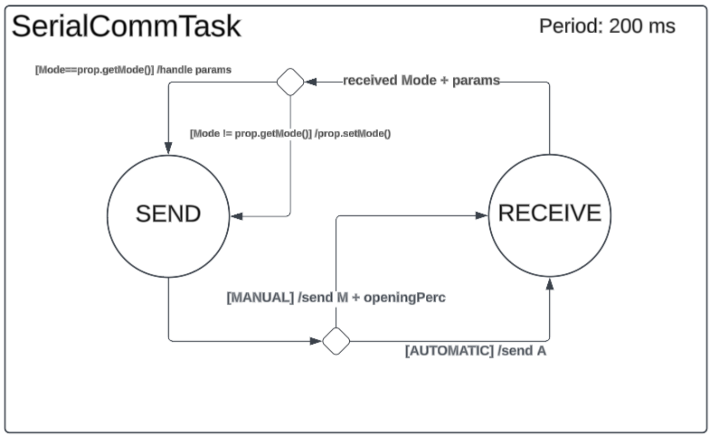

SerialCommTask
Period: 200 ms
[Mode==prop.getMode()] /handle params
received Mode + params
[Mode != prop.getMode()] /prop.setMode()
SEND
RECEIVE
[MANUAL] /send M + openingPerc
[AUTOMATIC] /send A

Figure 9: State diagram of SerialCommTask

4 Dashboard subsystem (Frontend/web app on the PC)

Main Classes

Main

The **Main** class represents the entry point of the program. It handles configuring the host and port for the connection, starting a dedicated event loop for asynchronous operations, and initializing the graphical dashboard.

Connection

The **Connection** class manages communication with a remote server through asynchronous HTTP requests. It provides methods to send data (POST), retrieve data (GET), and update modes or alarms on the server.

TemperatureDashboard

The **TemperatureDashboard** class implements the graphical interface of the monitoring system. It uses libraries such as `tkinter` and `matplotlib` to display dynamic graphs and update real-time data. It also includes buttons to interact with the system, such as changing modes or stopping alarms.

How the HTTP Connection Works

The HTTP connection is managed by the **Connection** class, which uses the `aiohttp` library to perform asynchronous requests to the server. This approach keeps the program responsive during network operations, avoiding delays caused by latency or other network issues.

POST Requests

POST requests are used to send data to the server. For example:

- **Sending temperature data**: The `post_data` method generates random values to simulate temperature readings and sends them to the server via a POST request.
- **Updating mode**: The `post_mode` method sends a new operating mode to the server.
- **Handling alarms**: The `post_alarm` method sends a request to stop an active alarm.

GET Requests

GET requests are used to retrieve data from the server. For example:

- **Retrieving temperature data**: The `get_data` method fetches temperature data stored on the server. This data is then processed and displayed in the graphical dashboard.

Asynchronicity and Event Loop

All HTTP operations are asynchronous, meaning they do not block the program's execution while waiting for a response from the server. This is made possible by the `asyncio` library, which manages a dedicated event loop. The event loop allows multiple tasks to run concurrently, improving the program's efficiency.

Data Format

The data exchanged between the client and the server is in JSON (*JavaScript Object Notation*) format. This format is lightweight and easy to interpret for both the client and the server. For example:

- A JSON object sent via POST may contain fields such as `value` (temperature value) and `place` (measurement location).
- A JSON object received via GET may contain a list of records, each with fields such as `val` (temperature value), `tst` (timestamp), and `opn` (opening level).