KAZAM ASSIGNMENT

Name: Ritika Srivastava

Registration Number: 21BEC10011

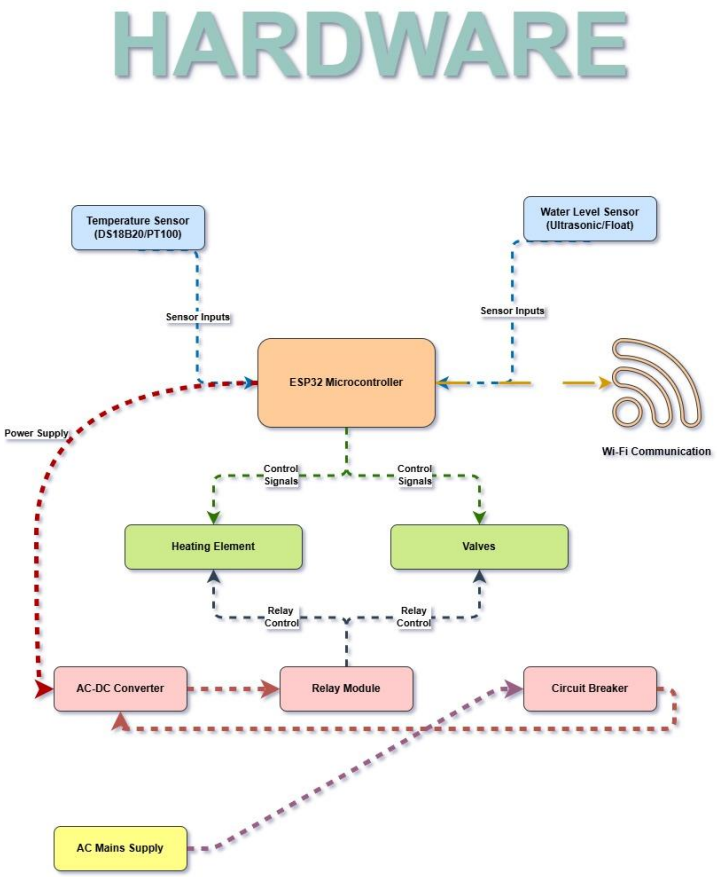
Part 1: Smart Water Heater

The aim of this project is to conceptualize and design a smart water heater system that incorporates advanced features such as energy efficiency, precise measurement, safety mechanisms, and remote operability. The system integrates hardware, firmware, and software components for seamless functionality and user convenience.

Hardware Design

Block Diagram

Below is the block diagram that visually represents the system's workflow:



The hardware architecture of the smart water heater is illustrated in the **Hardware Design Diagram**. It comprises the following components:

- **Sensors**:
  - **Temperature Sensor (DS18B20/PT100)**: Measures water temperature.
  - **Water Level Sensor (Ultrasonic/Float)**: Monitors the water level.
- **ESP32 Microcontroller**:
  - Acts as the central processing unit.
  - Collects data from sensors and controls actuators.
  - Enables Wi-Fi communication for remote operability.
- **Actuators**:
  - **Heating Element**: Controlled to maintain the desired temperature.
  - **Valves**: Regulate water flow and ensure system safety.
- **Power Supply System**:
  - **AC-DC Converter**: Converts AC mains to DC for low-voltage components.
  - **Relay Module**: Switches high-power devices like the heating element and valves.
  - **Circuit Breaker**: Provides overcurrent protection.

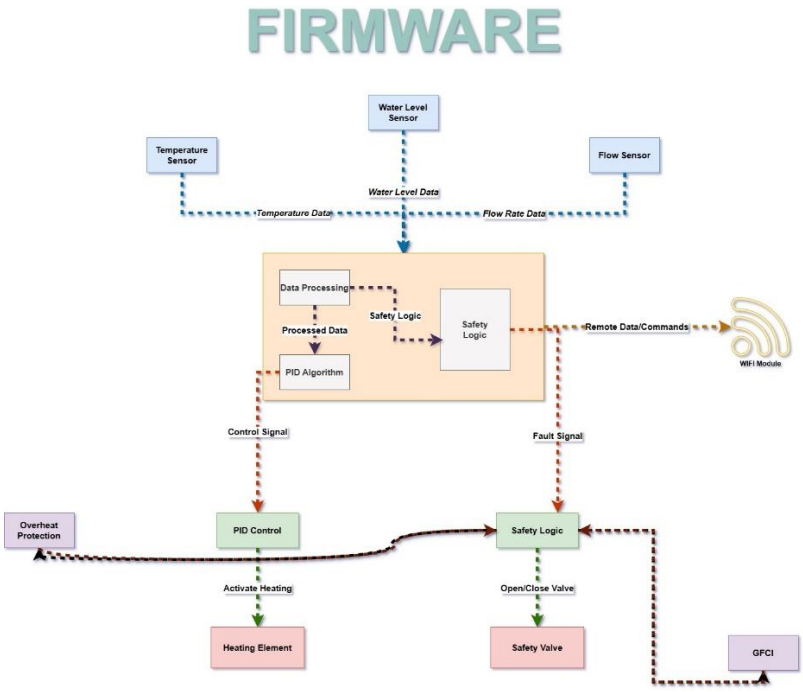
Key Interactions:

- Sensors send input signals to the ESP32 microcontroller.
- The microcontroller processes the data and sends control signals to actuators.
- The Wi-Fi module facilitates remote monitoring and control.

Firmware Design

Block Diagram

Below is the block diagram that visually represents the system's workflow:



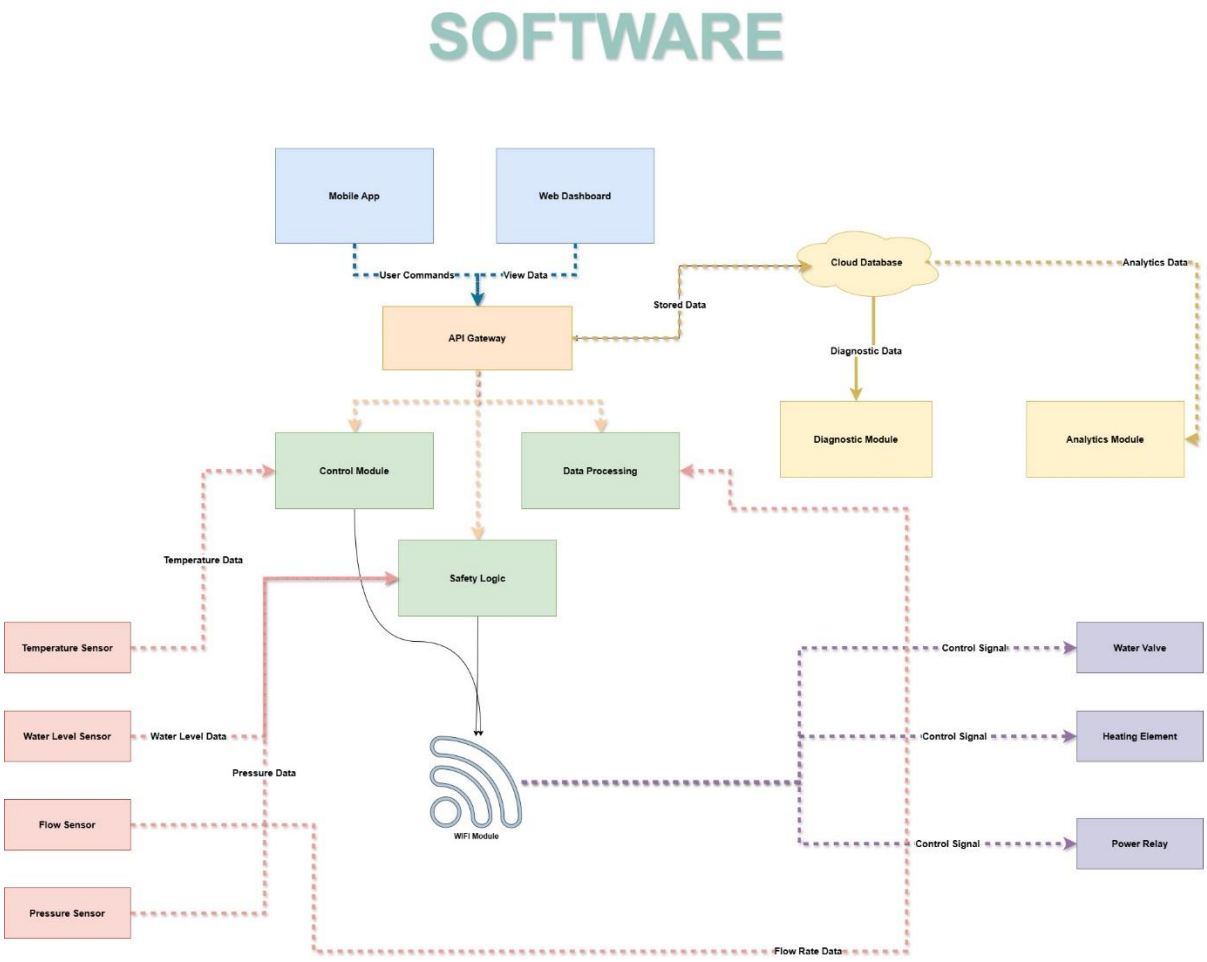
The firmware architecture is depicted in the **Firmware Design Diagram**. It includes:

- **Data Processing**:
  - Processes raw input from sensors (e.g., temperature and water level).
- **PID Algorithm**:
  - Ensures precise control of the heating element for energy efficiency.
- **Safety Logic**:
  - Implements safety mechanisms such as overheat protection and fault detection.
- **Communication**:
  - The microcontroller communicates with external systems via a Wi-Fi module.
- **Control Flow**:
  - Sensor data is processed and fed into the PID algorithm.
  - Control signals are sent to the heating element and valves.
  - Fault signals trigger safety mechanisms and disable the system if necessary.

Software Design

Block Diagram

Below is the block diagram that visually represents the system's workflow:



The software architecture is outlined in the **Software Design Diagram**, emphasizing user interaction, remote operability, and backend integration:

- **User Interfaces**:
  - **Mobile App**: Allows users to set preferences and monitor system status.
  - **Web Dashboard**: Provides advanced monitoring and analytics.
- **Cloud Connectivity**:
  - **API Gateway**: Facilitates communication between the hardware and the cloud.
  - **Cloud Database**: Stores data for analytics and diagnostics.
- **Backend Modules**:
  - **Control Module**: Manages user commands and system operations.
  - **Data Processing**: Analyzes sensor data for actionable insights.
  - **Diagnostic Module**: Detects faults and logs system performance.
  - **Analytics Module**: Provides insights on energy efficiency and usage patterns.

Key Interactions:

- Sensor data is processed locally and transmitted to the cloud for storage and analysis.
- Users interact with the system via the app or dashboard, sending commands through the API Gateway.
- Diagnostic and analytics data ensure optimal performance and system reliability.

Conclusion

The integration of the hardware, firmware, and software components enables the smart water heater to deliver energy-efficient, safe, and user-friendly functionality. This document highlights the comprehensive design approach, ensuring a robust and future-ready system.

Part 2: AC Load Bank

Code for the AC Load Bank

```
import matplotlib.pyplot as plt
import matplotlib.patches as patches

# Load bank steps
load_steps = [5000, 3000, 2000, 2000, 1000, 1000, 500, 500, 200, 200,
100, 100]

def calculate_relays(setpoint):
```

```
    relay_states = [False] * len(load_steps)
    for i in range(len(load_steps)):
        while setpoint >= load_steps[i]:
            setpoint -= load_steps[i]
            relay_states[i] = True

    if setpoint == 0:
        return relay_states
    else:
        return None  # in case setpoint not achievable

def visualize_circuit(relay_states):
    fig, ax = plt.subplots(figsize=(10, 6))
    ax.set_xlim(0, 10)
    ax.set_ylim(0, len(load_steps) + 2)
    ax.axis("off")

    for i, state in enumerate(relay_states):
        # Drawing the relay as a rectangle
        relay_color = "green" if state else "gray"
        relay = patches.Rectangle((1, len(load_steps) - i), 1, 0.8,
color=relay_color, edgecolor="black")
        ax.add_patch(relay)

        # Drawing the load step
        load_text = f"{load_steps[i]}W"
        ax.text(2.5, len(load_steps) - i + 0.4, load_text, va="center",
ha="center", fontsize=10)

        # Drawing the connecting lines
        ax.plot([2, 4], [len(load_steps) - i + 0.4, len(load_steps) - i
+ 0.4], color="black")
        ax.plot([4, 5], [len(load_steps) - i + 0.4, len(load_steps) - i
+ 0.4], color="black", linestyle="--")

    # Title and legend
    ax.text(0.5, len(load_steps) + 1.5, "Circuit Diagram: Relay
States", fontsize=14, fontweight="bold")
    ax.text(6, len(load_steps) - 0.5, "Green = ON", color="green",
fontsize=12)
    ax.text(6, len(load_steps) - 1.5, "Gray = OFF", color="gray",
fontsize=12)

    plt.show()

def main():
    print("Enter the load setpoint in watts (positive integer):")
    try:
```

```
        setpoint = int(input())
        if setpoint <= 0:
            print("Error: Invalid setpoint. Must be a positive
integer.")
            return

        relay_states = calculate_relays(setpoint)
        if relay_states:
            print("Setpoint achievable. Relays activated:")
            for i, state in enumerate(relay_states):
                if state:
                    print(f"Relay {i + 1} ON (Power:
{load_steps[i]}W)")

            # Visualizing the circuit using blocks
            visualize_circuit(relay_states)
        else:
            print("Error: Setpoint not achievable with available load
steps.")
    except ValueError:
        print("Error: Invalid input. Please enter a positive integer.")

if __name__ == "__main__":
    main()
```

Objective

The project aims to design and implement a program to control an AC load bank consisting of discrete load steps. The load bank is controlled via relays, with the primary objective of achieving a specific load setpoint (in watts) received through UART communication. The system ensures:

- The total power matches the input setpoint exactly (if achievable).
- The minimum number of relays is activated to optimize power usage and reduce relay wear.
- Error handling is in place for unachievable setpoints.

System Overview

The load bank consists of 12 discrete steps:

- 100W, 100W, 200W, 200W, 500W, 500W, 1000W, 1000W, 2000W, 2000W, 3000W, 5000W

Each step is individually connected to a relay. The relays are controlled by a microcontroller, which receives load setpoints via UART.

System Overview

The load bank consists of 12 discrete steps:

- 100W, 100W, 200W, 200W, 500W, 500W, 1000W, 1000W, 2000W, 2000W, 3000W, 5000W

Each step is individually connected to a relay. The relays are controlled by a microcontroller, which receives load setpoints via UART.

Implementation Details

3.1 Input Handling:

- The setpoint is a positive integer (in watts) received via UART communication.
- The program validates the setpoint to ensure it is a valid, positive integer.

3.2 Relay Control Logic:

- A greedy algorithm calculates the relay states needed to achieve the setpoint.
- Relays corresponding to larger load steps are prioritized to minimize the number of relays turned on.
- If the setpoint cannot be achieved, an error message is returned.

3.3 Communication:

- UART is used for communication between the user/microcontroller and the program.
- Relay states (ON/OFF) are sent back via UART for user feedback.

3.4 Visualization:

- A visual representation of the relay states is generated, showing which relays are ON (green) and OFF (gray), along with the corresponding load step sizes.

System Workflow

1. Receive load setpoint via UART.
2. Validate the input setpoint.
3. Calculate the relay states using the greedy algorithm.
4. Send relay states back via UART.
5. Visualize the relay states for debugging or demonstration purposes.
6. If the setpoint cannot be achieved, send an error message via UART.

Algorithm

Input: Load steps, setpoint **Output:** Relay states (ON/OFF)

- Initialize all relay states to OFF.
- Traverse the load steps (starting from the largest step):
    - While the setpoint >= load step, subtract the load step from the setpoint and turn the relay ON.
- If setpoint becomes 0, relay states are returned.
- If setpoint is not 0, return an error message (setpoint not achievable).

Result

Example: Setpoint = 3200W

- Active relays: 3000W (Relay 2) and 200W (Relay 9)
- Inactive relays: All others

The circuit diagram (visualization) illustrates this by showing green rectangles for active relays and gray rectangles for inactive ones.

Conclusion

This project successfully demonstrates an efficient method for controlling an AC load bank using relays. The system ensures minimal relay activation, optimal power usage, and robust error handling for invalid setpoints. The combination of UART communication and visualization provides an interactive and practical solution for load bank control.