G7: Mong Phan, Nadim Ahmed, Phuong Ta, Xuan Dang

Smart Plug Project

Third-year IoT Project Report

School of ICT

Metropolia University of Applied Sciences

23 March, 2025

Abstract

Authors:	Mong Phan, Nadim Ahmed, Phuong Ta, Xuan Dang
Title:	Smart Plug Project
Number of Pages:	61 pages + 4 appendices
Date:	23 March, 2025
Degree:	Bachelor of Engineering
Degree Programme:	Information Technology
Specialisation option:	Smart IoT Systems
Instructors:	Joseph Hotchkiss, Project Engineer Saana Vallius, Senior Lecturer

This project details the development of an IoT-based smart plug designed for remote monitoring and control of electrical appliances, with a focus on enhancing energy efficiency and user convenience. Built around the Raspberry Pi Pico W microcontroller, the system integrates FreeRTOS for real-time task management, MQTT for secure cloud communication, and an ACS712 current sensor for accurate power monitoring. Key features include remote on/off functionality, real-time energy tracking, user-defined power limits, and automated scheduling, all accessible through a customizable front-end interface and supported by a robust back-end server. Addressing limitations in existing smart plugs such as the absence of onboard displays and adjustable power thresholds, this project delivers a practical solution with advanced capabilities. The report covers the design, implementation, and testing phases, demonstrating reliable performance and offering potential for future enhancements like broader smart home integration and improved interface options.

Keywords: Raspberry Pi Pico W, FreeRTOS, MQTT, Current Sensor, IoT, Smart Plug, Energy Efficiency

Version history

Ver	Description	Date	Author(s)
1.0	Document created	2/2/2025	Mong Phan
1.1	Document outline reviewed	2/11/2025	Saana Vallius
1.2	Document finished	3/23/2025	Mong Phan Nadim Ahmed Xuan Dang Phuong Ta

Contents

1 Introduction

The Smart Plug Project is an IoT-based embedded system developed to provide users with remote monitoring and control over electrical appliances, promoting energy efficiency, convenience, and sustainability. Designed as a third-year project within the School of ICT at Metropolia University of Applied Sciences, this initiative integrates a Raspberry Pi Pico W microcontroller with wireless communication protocols, real-time operating systems, and cloud-based services to deliver a versatile and user-friendly smart plug. The system enables users to remotely toggle devices on or off, monitor energy consumption in real-time, set power consumption limits, and schedule automated operations, all through an intuitive interface supported by a cloud platform. By addressing shortcomings in existing commercial smart plugs such as limited feedback mechanisms, inflexible interfaces, and fixed power thresholds this project offers a practical solution with enhanced functionality.

The primary goal is to deliver a smart plug that provides users with precise control and actionable insights into their energy usage, reducing waste and optimizing appliance operation. This is achieved using embedded firmware developed with FreeRTOS, MQTT communication through the HiveMQ broker, and accurate current sensing with the ACS712 sensor, all supported by a custom hardware platform. The system overcomes common limitations in existing solutions, such as fixed power thresholds and reliance on external apps, by incorporating a local display and adjustable settings.

This report details the project's development process, beginning with the materials and methods used in Section 2. Section 3 provides an analysis of current smart plug shortcomings, while Section 4 covers the theoretical foundations, including MQTT, current sensing, and power consumption principles. Section 5 describes the implementation across local firmware, front-end, and back-end components, followed by testing and validation in Section 6. Section 7 concludes with reflections on the project's success and possibilities for future improvements, such as expanded compatibility with smart home ecosystems.

2 Methods and Material

This section outlines the materials, tools, and methods used to design, build, and test the smart plug, which enables remote control of devices with features such as on/off switching, power consumption limiting, timer scheduling, and power consumption reporting.

2.1 Material

This subsection lists and describes all hardware components, software tools, and other materials used in the development of the smart plug. Each component's role in the system is briefly explained.

The following hardware components were used in the design and construction of the smart plug:

- **Raspberry Pi Pico W**: A RP2040 based-microcontroller development board with built-in Wi-Fi and bluetooth capabilities. It serves as the central processing unit, handling all control logic, data processing, wireless communication, and interfacing with other components.

- **Color LCD**: A 1.54-inch color LCD with a resolution of 240x240 pixels, driven by the ST7789 controller and interfaced via SPI. It displays real-time information, including device's socket status, real-time clock (RTC) backup battery percentage, Wi-Fi connection status, power consumption, clock, scheduler, load voltage, current, and power, power setpoint, and QR code for Wi-Fi setup and device management.

- **Current Sensor**: ACS712-05B current sensor, capable of measuring currents up to 5A with a sensitivity of 185 mV/A. It measures the current drawn by the connected device to calculate power consumption and enforce power limits.

- **EEPROM**: STMicroelectronics M24256-BRMN6TP, a non-volatile memory chip with a storage capacity of 256 Kbits and interfaced via I2C. It stores Wi-Fi credentials, timer schedules, power setpoint, and accumulated power consumption, ensuring persistence of data during power outages.

- **RTC (Real-Time Clock)**: Microchip MCP79410-I/MS, a highly accurate real-time clock with an I2C interface and battery backup. It maintains accurate time for clock and scheduling functionality, even during power outages.

- **Hardware Over-Current Protection (OCP):** Unlike software OCP, which calculates instantaneous power consumption based on voltage and current readings from the ADC, hardware OCP relies on electronic components to respond quickly to current limits. It is designed to turn off the device's socket when the load reaches 4.5A. The OCP circuit consists of a comparator and a D-type flip-flop.

- **Mandatory Electronic Components**: Includes a 12V relay for switching DC loads, resistors, capacitors, mosfets, voltage regulators, connectors, and other passive components (see Appendix 3 for a detailed bill of materials). These components facilitate circuit operation, signal conditioning, power regulation, and safe switching of the connected device.

- **Printed Circuit Board (PCB)**: A custom-designed PCB, provides a compact and reliable platform for integrating all hardware components.

- **Power Supply:** A 12 VDC, 5 A power adapter serves as the main power source for the load and is stepped down to 5 V to supply power to the Pico W and all other components.

- **Enclosure**: A 3D-printed ABS plastic enclosure. It houses the smart plug components, ensuring safety, durability, and user-friendly design.

2.2 Methods

This subsection describes the step-by-step process of designing, building, and testing the smart plug. The process is divided into hardware design, software implementation, integration, and testing phases.

The hardware design process involved creating a schematic, designing a PCB, and assembling the components. The schematic was designed to integrate the Raspberry Pi Pico W with the 1.54" SPI LCD, current sensor, I2C EEPROM, I2C RTC, relay, and other electronic components. The schematic was created using EasyEDA. The PCB layout was designed to ensure compact component placement, adequate trace widths for high-current paths, and proper thermal management. The PCB was fabricated by JLCPCB. The final assembled board was carefully tested to ensure that each function worked properly.

The software implementation involved developing firmware for the Pico W, setting up remote communication with HiveMQ, and designing a user interface. The firmware was developed in C using FreeRTOS and the Pico SDK. The firmware architecture included several tasks, with inter-task communication managed via FreeRTOS queues and semaphores:

- **EEPROM Task:** Stored and retrieved user settings and data, such as timer schedules and power limits, using the I2C EEPROM.
- **GPIO Output Task:** Controlled the relay to enable or disable the device's socket, played sounds on a buzzer to indicate actions, and controlled the LED state to signal device status.
- **GPIO Input Task:** Listened to button inputs for user interactions (e.g., manual on/off control) and the OCP signal for over-current detection.
- **Internet Task:** Established a Wi-Fi connection using the Pico W's built-in CYW43439 Wi-Fi module and managed MQTT-related activities, including

publishing device status to specific topics (e.g., "smart_plug/post_wh") and subscribing to command topics (e.g., "timer").

- **ADC Task:** Monitored the RTC backup battery voltage and measured the load's voltage and current to calculate power consumption.

- **Buzzer Task:** Played different sounds for various activities, such as button presses, socket on/off states, and OCP triggers, using distinct tone patterns for user feedback.

- **Display Task:** Managed the color LCD to display the device state (e.g., on/off), schedule (e.g., timer settings), load statistics (e.g., real-time current, voltage, and power consumption), and QR code (e.g., for Wi-Fi setup or device management).

- **RTC Task:** Maintained the clock using the I2C RTC and triggered alarms for specific timers, such as user-defined schedules for turning the socket on or off.

The smart plug was configured to connect to the HiveMQ cloud-based MQTT broker for remote control. MQTT topics were defined for publishing data (e.g., on/off state, power consumption, acknowledgement to received commands) and subscribing to remote commands (e.g., turn on/off, set timers, set power limits). Security measures, such as TLS encryption, were implemented to ensure secure communication.

The hardware and software were integrated and tested to ensure the smart plug functioned as intended. The firmware was uploaded to the Pico W, and initial tests were conducted to verify communication between the Pico W and all peripherals (LCD, current sensor, EEPROM, RTC, buzzer, relay) and cloud communication as well. Integration challenges, such as accuracy in current sensor readings, and shared I2C bus 0 between EEPROM and RTC, were resolved by software filter in the ADC function, and correct definition of I2C instants. The following functional tests were performed:

- **Remote Control:** Verified the ability to turn the device on/off, set power setpoint, and set timers remotely via HiveMQ web client. Verified data is published to HiveMQ web client correctly.

- **Power Consumption Limiting:** Ensured the smart plug could detect and limit power consumption based on user-defined and hardware designed thresholds.

- **Timer Functionality (RTC):** Confirmed accurate scheduling of on/off events using the RTC with ability to add and remove timers.

- **LCD Display:** Validated the accuracy of information displayed on the LCD and refresh rate to any changes.

- **EEPROM Storage:** Tested the persistence of settings and data during power outages.

- **ADC Functionalities:** Verified the accuracy of ADC reading of RTC backup battery, load's voltage and current.

- **OTA Functionality:** Verified the device could perform firmware update remotely.

3 Current State Analysis

This section examines the shortcomings of the current state of the existing commercial smart plug to support the development of an enhanced smart plug. The analysis focuses on display capabilities, user interface customization, power limiting features, and the educational potential for students in applying Embedded IoT concepts to a practical end-to-end (E2E) solution.

3.1 Analysis of Existing Smart Plugs

- **Lack of Colorful and Large LCD Display:** Many popular commercial smart plugs, such as TP-Link Tapo, Deltaco, and Qnect, lack displays entirely, providing limited feedback to users and omitting detailed information such as real-time power consumption or device schedules. This absence of a colorful and large LCD display prevents users from accessing essential data (e.g., voltage, current, power usage, and timer settings) directly on the device, necessitating reliance on smartphone apps or external interfaces.

- **Inability to Customize User GUI:** Existing smart plugs typically feature fixed graphical user interfaces (GUIs) through mobile apps with predefined layouts. For instance, the TP-Link Tapo app does not allow users to tailor the display or control options to their specific needs. This limitation fails to accommodate diverse user preferences or support advanced applications, such as integrating custom IoT dashboards or adapting the interface for educational demonstrations.

- **Absence of Power Limiting Functionality:** Unlike the proposed design, most existing smart plugs do not enable users to set specific power limits, particularly in scenarios involving heavy loads paired with weak transmission lines. Their over-current protection (OCP), if present, remains fixed - for example, at 10 A in the Tapo P100 model - and cannot be adjusted to meet user-specific requirements.

3.2 Educational Context and Identified Need

Beyond addressing the technical deficiencies outlined in the previous subsections, this project serves a critical educational purpose by enabling students at Metropolia University of Applied Sciences to apply Embedded IoT knowledge in a practical, real-world context. The Degree Programme in Information Technology emphasizes competencies such as microcontroller programming, sensor integration, and network communication, which are foundational to IoT development. This smart plug project provides a platform for students to synthesize these skills into a complete end-to-end (E2E) IoT solution, comprising hardware design, firmware development, and cloud integration. Such an approach aligns with the programme's objective of preparing graduating engineers for systematic work and the transition to professional practice, as highlighted in the curriculum's focus on hands-on, industry-relevant projects.

In contrast, existing IoT products, particularly smart plugs, fail to support this educational goal. While these devices are made for consumer use, offering smart features, they do not provide an open source for students to explore the full picture of IoT development. For instance, commercial smart plugs such as the TP-Link Tapo P100 or Deltaco SH-P01 are closed systems with proprietary firmware and hardware, limiting visibility into their internal workings. This opacity restricts students from experimenting with key IoT concepts, such as real-time data processing, custom protocol implementation, or hardware-software interfacing. Consequently, these products serve as end-user tools rather than educational platforms, offering little opportunity for hands-on learning beyond basic app-based operation.

The lack of transparency in commercial IoT solutions poses several specific barriers to student learning. First, proprietary firmware prevents students from modifying or analyzing the software architecture, a critical skill in embedded systems development. Second, the absence of onboard displays or customizable interfaces in these products limits students' ability to observe and manipulate real-time data interactions, such as power consumption metrics or network status, directly on the device. This restricts the development of debugging and optimization skills, which are essential for IoT engineers. Third, the fixed functionality of existing smart plugs, such as non-adjustable over-current protection (OCP), precludes experimentation with advanced features like

user-defined power limits, a capability this project incorporates to deepen understanding of power management principles.

The need this project addresses goes beyond just improving technology, it also tackles a gap in how IoT is taught and learned with existing products. By developing a smart plug with an open design, featuring a customizable GUI, a large color LCD, and adjustable power limits, this project empowers students to engage directly with the design, implementation, and testing phases of IoT development. This hands-on approach not only reinforces classroom learning but also fosters skills in problem-solving, critical analysis, and innovation, which are vital for engineering professionals. Furthermore, the project's emphasis on a complete E2E solution mirrors real-world engineering tasks, preparing students for industry challenges where they must integrate hardware, software, and network systems seamlessly. Thus, the analysis of existing smart plugs reveals a dual deficiency: their inadequacy as consumer tools with advanced features and their unsuitability as educational platforms, both of which this project seeks to rectify.

4 Theory

This section explores the theoretical principles related to the development of the smart plug, addressing the gaps identified in the Current State Analysis section. Relevant concepts include the Message Queuing Telemetry Transport (MQTT) protocol for network communication, current sensing techniques for load monitoring, and methods for calculating electricity power consumption. Additionally, the theory of firmware remote update over-the-air (OTA) is examined as a key mechanism for maintaining and enhancing device functionality. These theories provide the foundation for designing an IoT-enabled smart plug with enhanced display, customizable power limits, and educational utility.

4.1 Message Queuing Telemetry Transport (MQTT)

MQTT (Message Queuing Telemetry Transport) is a lightweight messaging protocol pivotal to Internet of Things (IoT) ecosystems. Designed for efficient data exchange in constrained environments, MQTT supports a range of applications by leveraging a flexible and robust communication framework. The analysis covers MQTT versions 3.1.1 and 5.0, detailing their architecture, features, and benefits based on authoritative sources.

Introduced by IBM in 1999 and later standardized by OASIS, MQTT utilizes a publish-subscribe paradigm where a central broker coordinates message flow between clients [1][2]. Publishers send messages to topics - logical channels like "smart_plug/current" - while subscribers receive updates from topics of interest via the broker, enabling scalable, decoupled communication [3]. Figure 1 depicts this structure, fundamental to both MQTT 3.1.1 and 5.0 [4].

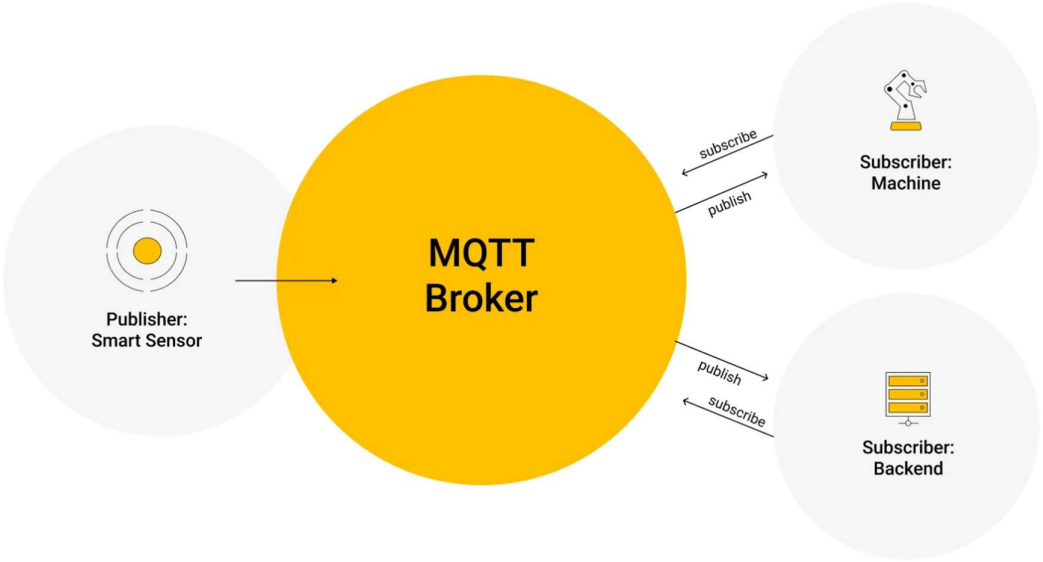


Figure 1: MQTT Publish/Subscribe Architecture [4].

As illustrated in Figure 1, the broker acts as a hub, relaying messages from publishers (e.g., a sensor) to subscribers (e.g., a monitoring app), facilitating real-time, asynchronous exchanges [4]. In MQTT's publish-subscribe model, data exchange revolves around topics, which serve as named channels for routing messages [3]. Publishers generate and send data - such as environmental readings or status updates - to a specific topic by creating a message payload and transmitting it to the broker, which acts as a central dispatcher [2]. For example, a publisher might send a temperature value to the topic "environment/temp." Subscribers, in turn, register their interest by subscribing to one or more topics via the broker, receiving all messages published to those topics [4]. The broker matches incoming messages to subscriptions and forwards them to relevant subscribers, ensuring efficient delivery without direct publisher-subscriber interaction [1]. This process supports dynamic communication, as subscribers can join or leave topics at any time, and multiple subscribers can receive the same message simultaneously, enhancing flexibility and scalability in IoT systems [3]. MQTT 3.1.1 established a minimal-overhead design with small packet headers (2 bytes fixed headers), optimized for low-bandwidth or unstable networks, while 5.0 introduces enhancements for greater functionality [5][6]. Key features include Quality of Service (QoS) levels, Security supports, Scalability, Persistent Sessions,

Lightweight and efficient, and Bidirectional communication, each contributing to MQTT's effectiveness [4].

- **Quality of Service (QoS) Levels:** MQTT's Quality of Service (QoS) levels ensure flexible message delivery tailored to application needs [3]. QoS 0 delivers messages once without confirmation, prioritizing speed for non-critical data like frequent updates, though loss is possible [2]. QoS 1 guarantees at least one delivery via a PUBACK acknowledgment, retransmitting if unconfirmed, suitable for essential commands where duplication is acceptable [5]. QoS 2 uses a four-step handshake (PUBLISH, PUBREC, PUBREL, PUBCOMP) to ensure exact delivery, ideal for critical operations, despite higher latency [6]. MQTT 5.0 upgrades the MQTT 3.1.1 with some adjustments like "Last Will and Testement", "TTL", and "Shared Subscriptions" [7].

- **Security Supports:** Security in MQTT is robust, protecting data across IoT networks [1]. Transport Layer Security (TLS) encrypts communications between clients and brokers, preventing eavesdropping and tampering, while authentication -via username-password pairs or certificates - restricts access to authorized entities [2]. MQTT 5.0 advances this with detailed error codes and client-specific feedback, enhancing troubleshooting compared to 3.1.1's basic mechanisms [5]. Brokers can enforce secure connections, ensuring confidentiality and integrity for sensitive data exchanges [3]. This layered approach mitigates risks in distributed systems, offering a significant advantage over unsecured protocols and supporting MQTT's adoption in privacy-sensitive applications like healthcare or industrial monitoring [1].

- **Scalability:** MQTT's scalability stems from its topic-based publish-subscribe model, enabling efficient handling of numerous clients [3]. Topics (e.g., "device/status") allow precise message routing, with wildcards - single-level (+) and multi-level (#) - supporting broad subscriptions (e.g., "device/+/status") [6]. The broker manages subscriptions and distributions centrally, decoupling publishers and subscribers, which allows systems to scale without direct connections [2]. This architecture supports dynamic growth, as new devices can join topics without reconfiguring existing clients [4]. MQTT 5.0's shared

subscriptions further enhance scalability by distributing messages across multiple subscribers, optimizing load in large-scale IoT deployments like smart cities [5].

- **Persistent Sessions:** Persistent sessions in MQTT maintain client state across disconnections, ensuring continuity [3]. When a client connects with a persistent session flag, the broker retains its subscriptions and undelivered QoS 1 or 2 messages during offline periods [6]. Upon reconnection, the client resumes without re-subscribing, receiving queued messages, which is crucial for unreliable networks [2]. MQTT 5.0 refines this with session expiry intervals, allowing sessions to terminate after a set time, freeing resources if clients remain offline [5]. This feature enhances reliability in intermittent environments, distinguishing MQTT from protocols requiring constant connectivity and supporting robust IoT operations [1].

- **Lightweight and Efficient:** MQTT's lightweight and efficient design optimizes resource use in constrained environments [4]. Its binary packet structure, with headers as small as 2 bytes, reduces bandwidth overhead compared to text-based protocols like HTTP [3]. This efficiency minimizes power consumption, making it ideal for battery-powered or low-processing devices [2]. Keep-alive pings maintain connections with minimal data, ensuring stability over unreliable networks [6]. MQTT 3.1.1 pioneered this approach, while 5.0 maintains it with added flexibility, such as message size limits [5]. This compactness enables rapid, energy-efficient communication, a key factor in MQTT's suitability for IoT applications requiring minimal resource footprints [1].

- **Bidirectional Communication:** MQTT supports bidirectional communication, allowing seamless data flow between clients [3]. Unlike unidirectional protocols, publishers can also subscribe, and subscribers can publish, enabling two-way interactions via the broker [2]. For instance, a device publishing sensor data to "sensor/data" can subscribe to "sensor/control" for commands, facilitating real-time feedback loops [4]. The broker's role as an intermediary ensures messages are routed efficiently in both directions, supporting dynamic exchanges [6]. This capability, consistent in 3.1.1 and 5.0, enhances MQTT's

versatility for interactive IoT systems, such as remote monitoring and control, where reciprocal communication is essential [1].

4.2 Current sensing (ACS712 family sensor)

ACS712 current sensor family, developed by Allegro MicroSystems, provides a widely used solution for precise AC and DC current measurement in various systems. Employing the Hall-effect principle, the ACS712 offers an economical, isolated method for current sensing, applicable in industrial, commercial, and communication contexts. This discussion outlines its operational mechanism, key characteristics, and typical uses, enhancing understanding of its role in electrical engineering applications.

The ACS712 integrates a linear Hall-effect sensor with a copper conduction path to measure current flow [8]. Current through the path (pins 1-2 to 3-4) generates a magnetic field proportional to its magnitude and direction, which the Hall sensor converts into a voltage output, centered at 2.5 V (for a 5 V supply) at zero current [8]. Figure 2, sourced from Allegro's datasheet, depicts this configuration within the sensor's SOIC8 package.

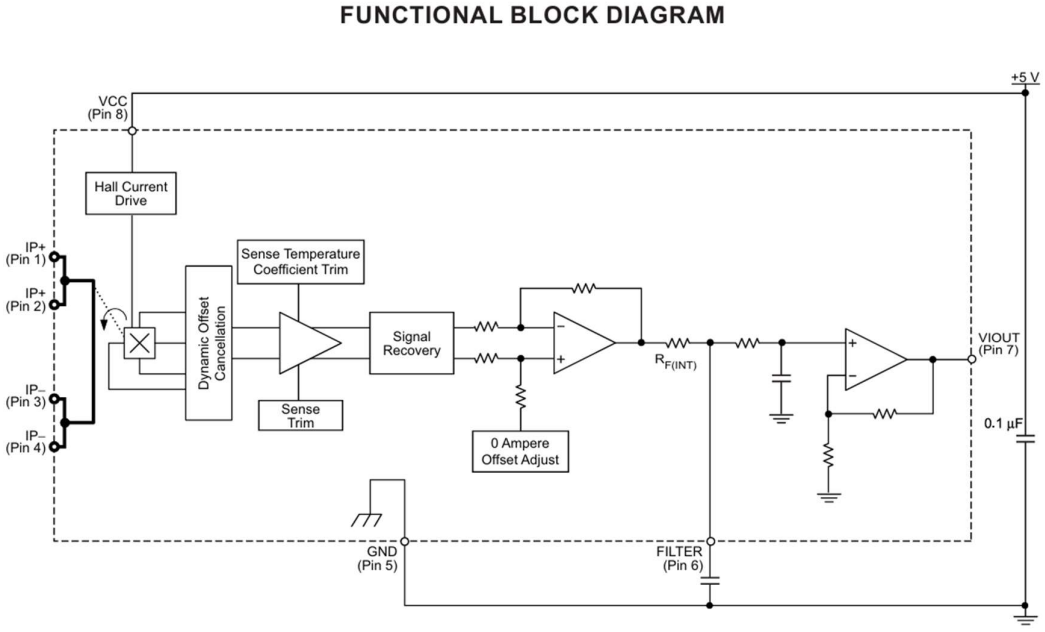


Figure 2: Functional block diagram of ACS712 [8]

As illustrated in Figure 2, the output voltage varies linearly with current -rising above 2.5 V for positive flow and dropping below for negative - offering sensitivities of 185 mV/A, 100 mV/A, or 66 mV/A for the 5 A, 20 A, and 30 A models, respectively [8]. With a 5 V supply, it achieves 2.1 kVRMS isolation, a 1.2 mΩ conduction path resistance, and ±1.5% accuracy at room temperature [8]. Bidirectional sensing, an 80 kHz bandwidth, and a compact design suit it for motor control, load management, and overcurrent protection applications [8].

4.3 Electricity Power Consumption and Calculation

Electricity power consumption and its calculation are critical for understanding energy usage in electrical systems. These principles apply to both direct current (DC) and alternating current (AC) circuits, with a particular emphasis on single-phase AC systems prevalent in domestic and industrial applications. The discussion draws on established electrical engineering concepts to clarify power types, measurement techniques, and their significance.

Power consumption represents the rate at which electrical energy is utilized, typically expressed in watts (W) [9]. In DC systems, power (P) is calculated as $P = V \times I$, where V is voltage and I is current. However, AC systems introduce complexity due to the sinusoidal variation of voltage and current, which may not align temporally [10]. This phase difference introduces the power factor (PF), defined as the cosine of the angle (φ) between voltage and current waveforms. The power factor, ranging from 0 to 1, quantifies efficiency, with 1 indicating perfect alignment (resistive loads) and lower values reflecting reactive effects from inductive or capacitive elements [9].

Three power types characterize AC systems: apparent power (S), real power (P), and reactive power (Q), measured in volt-amperes (VA), watts (W), and volt-amperes reactive (VAR), respectively [10]. Apparent power, given by $S = V_{rms} \times I_{rms}$ (where V_{rms} and I_{rms} are root mean square values), represents total power supplied. Real power, the energy actually consumed, is calculated as $P = S \times PF$, or $P = V_{rms} \times I_{rms} \times \cos(\varphi)$ [9]. Reactive power, which oscillates without being consumed, is $Q = S \times \sin(\varphi)$ [10]. Real power drives useful work (e.g., heating, motion), while apparent

power determines system capacity requirements, and reactive power affects voltage stability [9].

Measurement typically involves sensing current and voltage, often using devices like Hall-effect sensors for current and voltage dividers for potential [8]. For resistive loads (PF ≈ 1), P ≈ V × I suffices, but reactive loads require phase angle determination to compute PF accurately [10]. Energy consumption, derived as E = P × t (where t is time in hours), is expressed in watt-hours (Wh), providing a cumulative usage metric [9]. These calculations underpin efficient energy management and system design across electrical applications.

4.4 Firmware remote update Over-The-Air (OTA)

Over-the-air (OTA) updates replace the current microcontroller firmware with the new firmware. Embedded systems are usually flashed with the firmware once after testing and debugging the firmware. Mostly devices are deployed in places that are inaccessible by the developers or humans to change or update the firmware. In IoT devices it is more practical to update firmware over-the-air, it is a flexible, cost effective and efficient mechanism.[12]

OTA update mechanism can be implemented using multiple communication methods like bluetooth, internet or other radio communication. The simplest OTA workflow is notification, download, verify and update. Figure 3 shows the workflow of OTA update.[13]

Figure 3: Workflow of the OTA update

OTA update requires to download the new firmware and save it to the flash memory and needs to notify the bootloader to swap the binary during the next reboot. Second

stage bootloader needs to be implemented in case the first stage bootloader does not support OTA update. Section 4.5 explains the implementation of second stage bootloader and linker script in this project.[12]

The major challenges arise during implementation of OTA functionality to an embedded device are memory, communication protocol and security. As most often embedded devices consist of fixed and less amount of flash and ram memory, it is required to manage the memory, so it does not get out of memory during download. Most common communication protocols used for OTA are MQTT and HTTP. MQTT has less overhead, faster and consumes less ram memory in runtime, but packets might get loss if request and response communication method is not implemented correctly. On the other hand, HTTP method requires more ram memory cause of large header but also has less chance to packet loss. Security is the final major challenge in OTA implementation. Server needs to be trusted party it needs authentication, encryption and crc checking to ensure the new firmware is not corrupted.

4.5 Second Stage Bootloader

Bootloader is a software that resides on the read only memory. This software is executed every time the device gets reset. User does not have access to this layer of software. It is responsible for boot sequence [12]. RP2040 call this layer of software is bootrom. It performs the processor-controlled part of the boot sequence. RP2040 consist of hardware-controlled part of boot sequence that is completely referred to the voltage, power and clk pin related staff that is not directly related with the bootrom.

Bootrom has 16KB limited memory. It is responsible for processor 0 initialization and initializing flash, ram and other necessary library and functions. It gives booting options for application from flash. It also provides USB bootloader option to download application data or code directly to flash or ram. But this does not have built in functionality for OTA update [17].

Implementing the OTA update functionality requires to add second stage bootloader inherited from pico stage 2 bootloader. Raspberry pi provides this skeleton for the developer to implement their own version of bootloader. This helped developers to

implement Micro Python to program the RP2040. The main purpose of the second stage bootloader to implement the functionality to start the current application from different address point and also to be able to swape the application images after update.

It requires to make partition into to the flash memory to save multiple application binary and other necessary flags. The first application that is flashed in to the program for the first time will stay in separate memory location and another one that gets downloaded for update the application. It is also necessary to make changes into the linker script to make separate isolated memory partitions. Linker scripts are written in specialized scripting language to specify the format and layout of the final executable binary and to control the memory layout of the output file. Implementing bootloader requires enough time and deep knowledge of linker script. To implement OTA into this device we chose open sources bootloader called “Raspberry Pi Pico W FOTA Bootloader” by Jakub Zimnol. This bootloader consists of necessary functionality and linker script for bootloader and application. The Figure 4 shows the memory mapping in the linker script for application and bootloader.

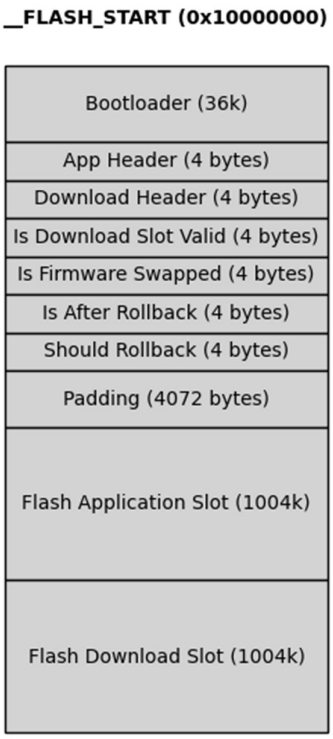


Figure 4: Memory mapping of second stage bootloader

The linker script divided the memory layout in several parts to save he current application and new application separately and to keep some of the flags to determine boot sequence from the new application or from old application. Application slots have been divided in equal two 1004KB memory sections also 36KB for second stage bootloader. Bootloader also have the basic functionality to rollback and firmware swapping mechanism and separated flag location in memory [15].

5 Implementation

The development and functionality of the Smart Plug system are detailed in this section, providing a comprehensive overview of its realization from concept to operational prototype. Emphasis is placed on the key algorithms and software components that constitute the system's core, enabling features such as real-time power monitoring, remote control, user-defined schedules, and power limiting. This implementation integrates hardware and software into a cohesive end-to-end solution, addressing both technical performance and educational objectives. To illustrate the system's architecture and interactions, a system diagram is presented in Figure 5, offering a visual representation of the components and their interconnections.

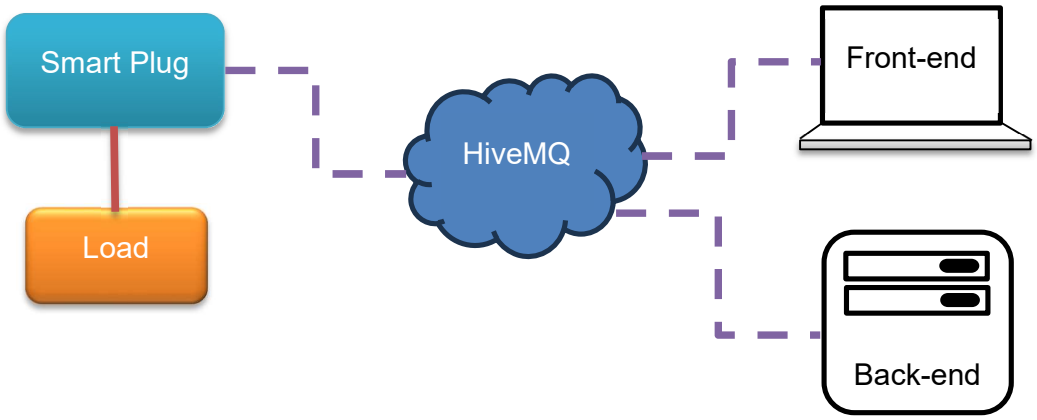


Figure 5: End-to-End Architecture of the Smart Plug System.

Figure 5 depicts the end-to-end architecture of the Smart Plug system, comprising four main components: the Smart Plug connected to the Load, the HiveMQ cloud broker, the Front-end application, and the Back-end service. The Smart Plug connects to the Load, managing power delivery, while communicating with the HiveMQ broker to facilitate remote interactions. The HiveMQ broker, a central messaging hub, enables data exchange between the Smart Plug and the Front-end/Back-end, which handles user interactions and system logic. This architecture ensures seamless integration of hardware control and cloud-based communication, supporting real-time monitoring and control functionalities.

5.1 Local System

The local system of the smart plug is implemented on the Raspberry Pi Pico W, leveraging FreeRTOS to manage real-time operations and ensure efficient resource utilization. FreeRTOS, a real-time operating system, provides a robust framework for multitasking, enabling the system to handle multiple concurrent processes with precise timing and synchronization. This is particularly advantageous for the smart plug, which requires simultaneous management of hardware interfaces, network communication, and user interactions in a resource-constrained environment. The system's firmware is structured into distinct tasks, each responsible for a specific function, ensuring modularity, maintainability, and scalability. These tasks communicate and synchronize using FreeRTOS mechanisms such as queues, semaphores, and event groups, facilitating seamless interaction across the system.

The local system is divided into several key tasks, each designed to handle a dedicated aspect of the smart plug's functionality. The Main Function initializes the system, setting up data structures, event groups, and tasks before starting the FreeRTOS scheduler. The EEPROM Task manages persistent storage, handling Wi-Fi credentials, power setpoints, timer schedules, and power consumption offline backup in the EEPROM. The Internet Task oversees MQTT communication with the HiveMQ broker, enabling remote control and monitoring. The ADC Task processes analog-to-digital conversions for current sensing, load voltage, and RTC battery measurements. The GPIO Input Task monitors input signals, such as user button presses and OCP signal, while the GPIO Output Task controls output devices like the relay for switching the socket, a bi-color LED for status indication, and a buzzer for events notification. The Real-Time Clock (RTC) Task manages timekeeping and timer schedules, ensuring accurate triggering of events. The Display Task updates the 1.54" SPI LCD with real-time data, such as power usage and schedules. The Buzzer Task handles audible alerts for events like buttons pressed or socket state changed, and the OTA Task facilitates over-the-air firmware updates for system maintenance. This task-based architecture ensures that each function operates independently yet collaboratively, supporting the project's goal of delivering a reliable, end-to-end IoT

solution. The following subsections provide detailed explanations of each task's implementation and operation.

- **Main function:** The firmware for the smart plug, developed using FreeRTOS on the Raspberry Pi Pico W, manages the system's operation through a structured main function. This function initializes the system's data structures, sets up event groups for task synchronization, creates tasks to handle various functionalities, and starts the FreeRTOS scheduler to manage task execution. The main function ensures that all components - such as hardware interfaces, communication protocols, and user interactions - are properly configured before entering an infinite loop, allowing the scheduler to run tasks indefinitely. Figure 6 illustrates the flowchart of this main function, detailing the sequence of operations.

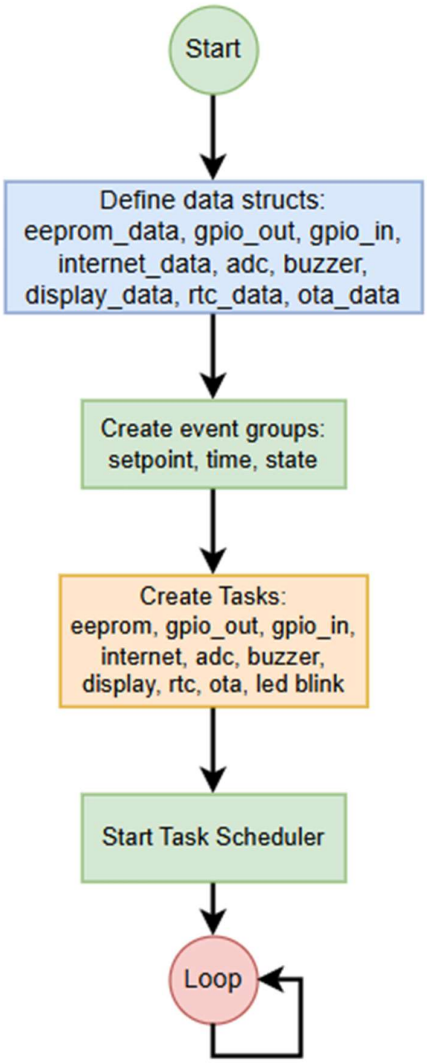


Figure 6: Flowchart of the Main Function for the Smart Plug Firmware.

As depicted in Figure 6, the main function begins with the "Start" point, representing the program's entry. The first step, "Define data structs," involves initializing key data structures: eeprom_data for persistent storage, gpio_out and gpio_in for GPIO pin configurations, internet_data for MQTT communication, adc for current sensing, load_voltage and rtc_battery for load voltage and RTC battery measurements, buzzer for alerts, display_data for LCD updates, rtc_data for timekeeping and timer triggers, and ota_data for over-the-air updates. This step establishes the foundation for communication channels among tasks. Next, "Create event groups" sets up synchronization mechanisms (setpoint, time, state) to coordinate task interactions, such as

disabling the socket when power limits are exceeded. The "Create Tasks" step then spawns individual FreeRTOS tasks - eeprom, gpio_out, gpio_in, internet, adc, buzzer, display, rtc, ota, and led_blink - each responsible for a specific function, such as reading sensors, managing MQTT messages, or updating the LCD. Finally, "Start Task Scheduler" activates the FreeRTOS scheduler, which manages task execution based on priorities and events, leading to an infinite "Loop" where the system operates continuously.

- **EEPROM Task:** The EEPROM Task is a critical component of the smart plug's firmware. This task manages the storage and retrieval of persistent data in the EEPROM (Electrically Erasable Programmable Read-Only Memory), ensuring that essential settings - such as Wi-Fi credentials, power setpoints, timer schedules, and Wh data - are retained across power cycles. The task also handles updates to these settings based on user inputs or system events, maintaining synchronization with other tasks like the Internet, RTC and Display Tasks via event bits and queues. Figure 7a and Figure 7b present the flowchart of the EEPROM Task, illustrating its operational logic through two main processes: initialization and runtime updates.

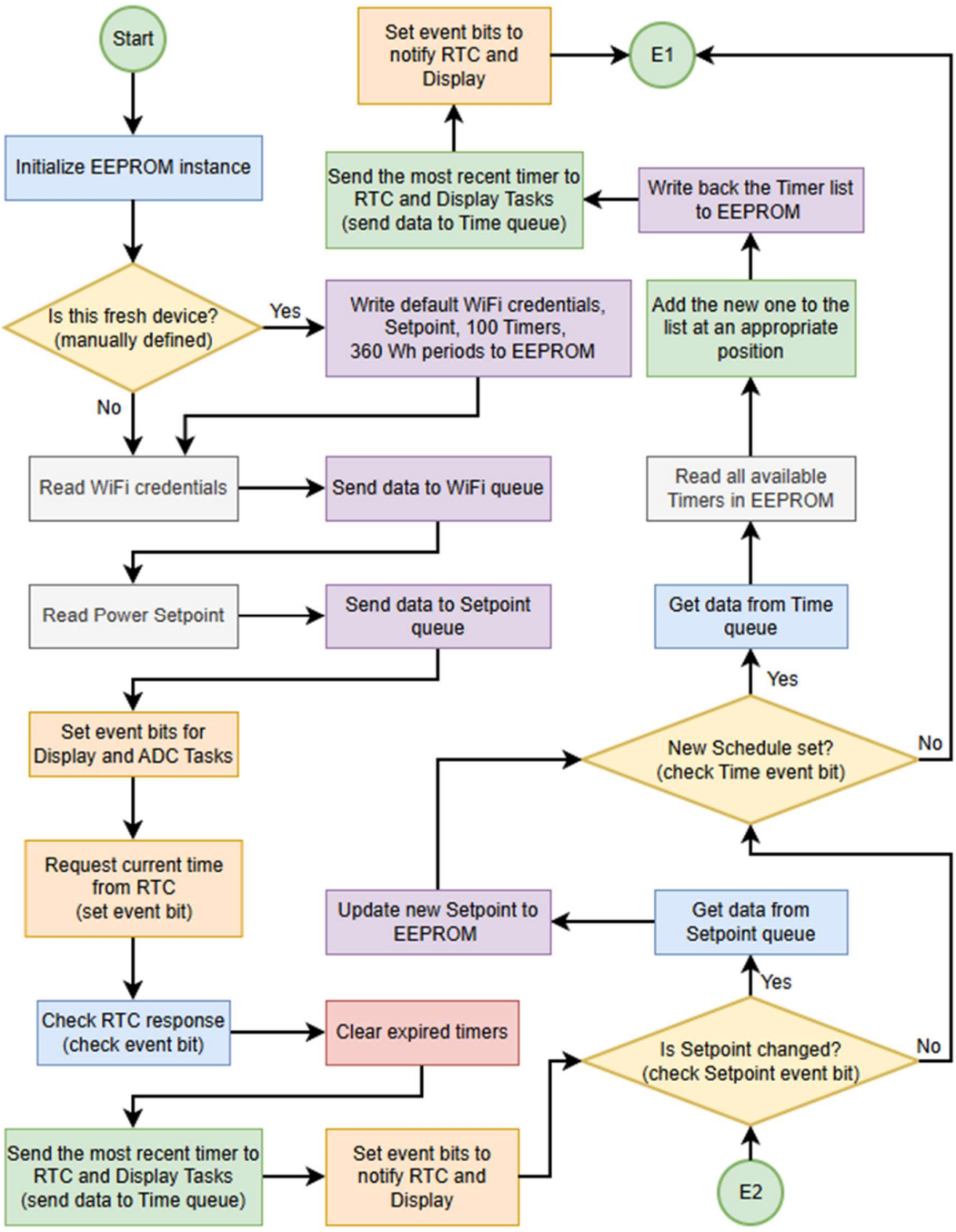


Figure 7a: Flowchart of the EEPROM Task for the Smart Plug Firmware.

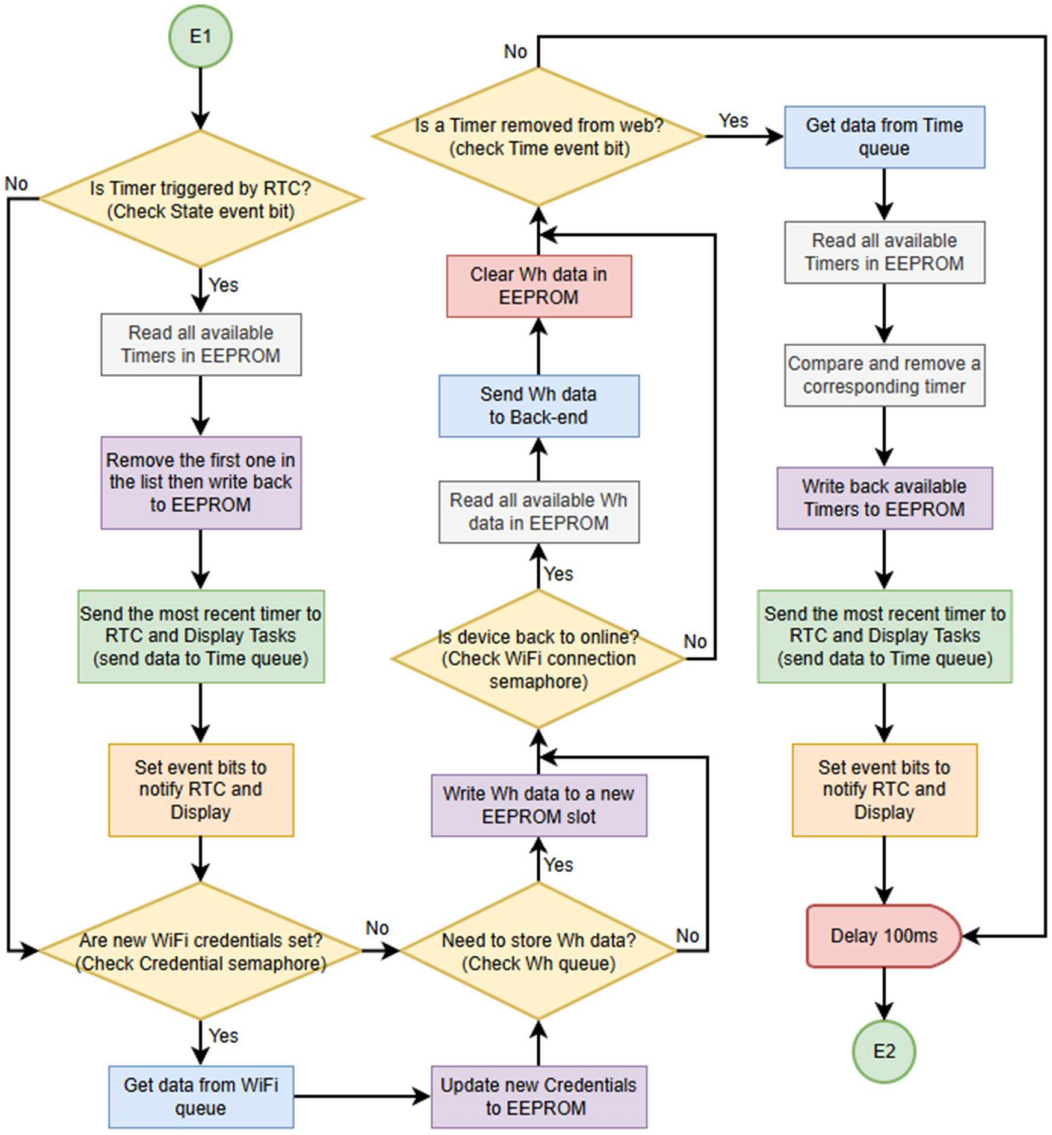


Figure 7b: Flowchart of the EEPROM Task for the Smart Plug Firmware.

As depicted in Figure 7a and Figure 7b, the EEPROM Task starts by initializing an EEPROM instance to interface with the EEPROM chip. The program then checks if the device is newly manufactured by verifying whether FRESH_BOAD is manually defined. If it is a new device, default Wi-Fi credentials, a setpoint value, 100 timer entries, and 360 slots of power consumption data (in Wh) are written to the EEPROM to clear any residual data in the new memory. Following this, the task retrieves the stored Wi-Fi credentials and power setpoint, sending them to the corresponding queues for other tasks to process. To manage existing timer schedules in the EEPROM, the task requests the current time

(hh:mm) from the RTC Task to identify and clear expired timers. It then sends the most recent timer in the list to the RTC and Display Tasks to set an alarm and display the next schedule, respectively. The program proceeds by checking if a new power setpoint has been set via the web application, updating the EEPROM with the new setpoint if available, or continuing to the next operation if not. If a user sets a new timer schedule, the task reads all existing timers from the EEPROM, combines them with the new timer to create an updated list, writes this list back to the EEPROM, and sends the most recent timer to the RTC and Display Tasks for processing. The program then checks the State event bit to determine if a timer has been triggered by the RTC; if set, the first timer in the list is removed, the list is updated, and the next timer is sent to the RTC and Display Tasks. Subsequently, the program verifies if new Wi-Fi credentials have been set (e.g., when the user connects the device to a new Wi-Fi network). If the Credential Semaphore is given, the new credentials are updated in the EEPROM. Next, the Task checks if the Wi-Fi state has changed from offline to online; if so, it sends the stored Wh data in the EEPROM to the backend and clears the Wh data from the EEPROM. If a user removes a timer via the web application, the EEPROM Task is notified through a Time event bit, prompting the program to read all available timers in the EEPROM, remove the specified timer, write the updated list back to the EEPROM, and send the most recent timer to the RTC and Display Tasks. The task then delays for 100 ms before repeating the loop.

- **Internet Task:** The Internet Task, a key component of the smart plug's IoT system, manages network connectivity and MQTT communication with the HiveMQ broker. This task ensures the smart plug can connect to Wi-Fi, synchronize time via NTP, exchange data with the web application, and handle remote control and monitoring functions. It interacts with other tasks, such as the EEPROM, Display, and RTC Tasks, using queues, semaphores, and event bits to maintain system synchronization. Figure 8a and figure 8b illustrate the complete flowchart of the Internet Task, detailing its operational flow from initialization to runtime management.

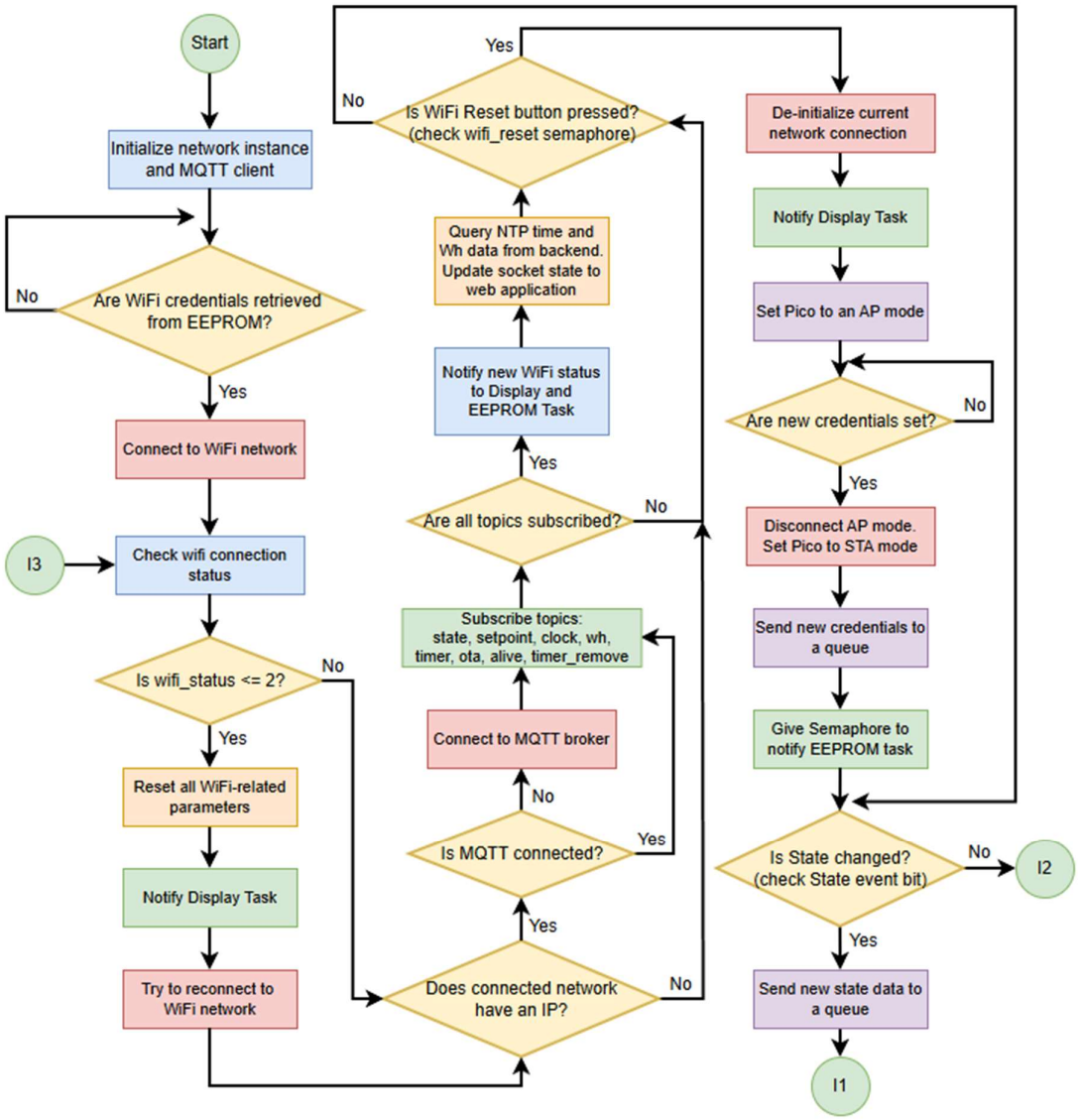


Figure 8a: Flowchart of the Internet Task for the Smart Plug Firmware.

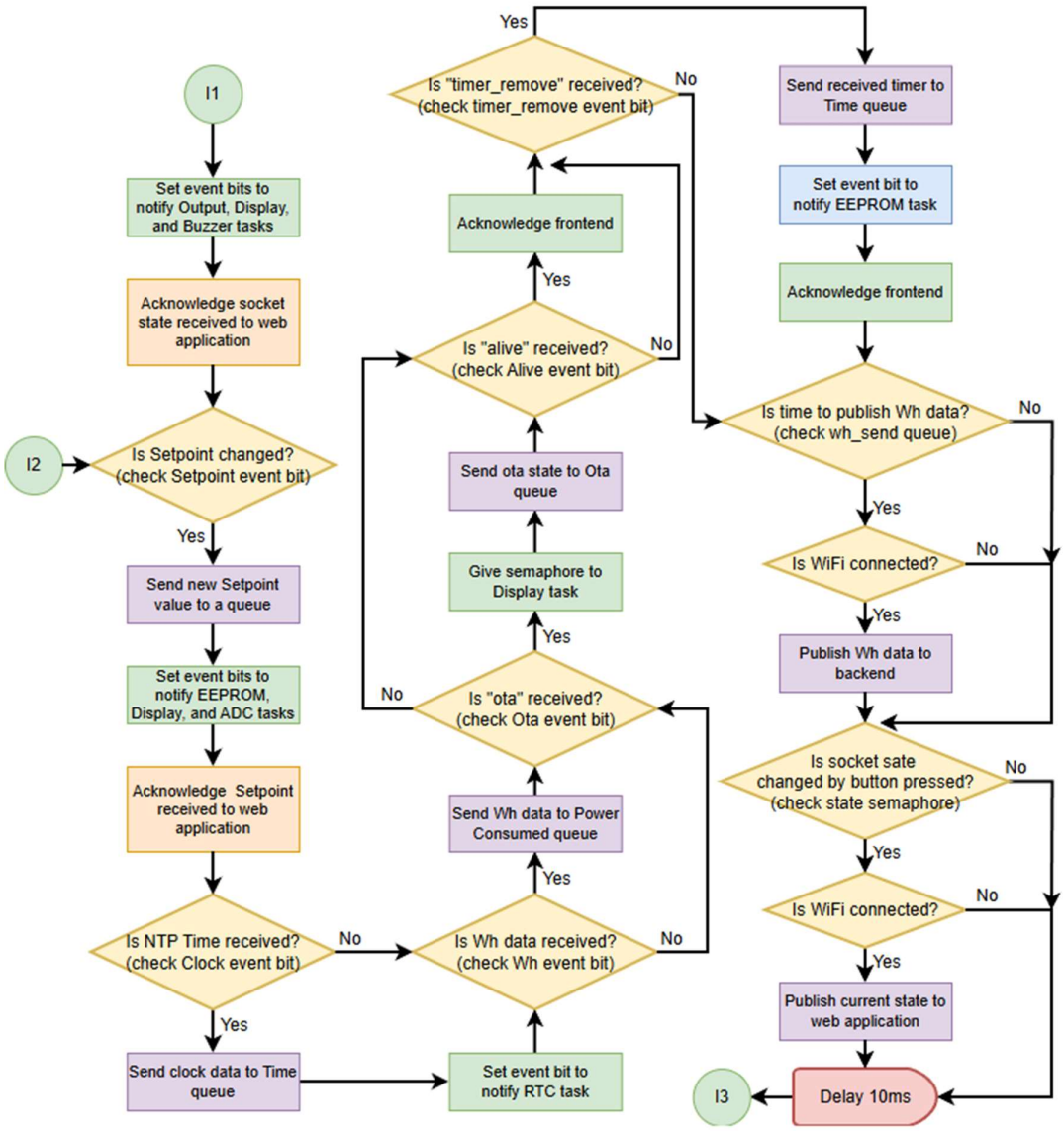


Figure 8b: Flowchart of the Internet Task for the Smart Plug Firmware.

As depicted in Figure 8a and figure 8b, the Internet Task starts by initializing the network instance and MQTT client to establish communication capabilities on the Raspberry Pi Pico W. The program then retrieves the Wi-Fi credentials from the EEPROM Task to connect to the network. The program checks the network connection status to determine the appropriate action. If the status is 2 or lower, indicating a failure to connect to the Wi-Fi network, the program resets all Wi-Fi-related parameters and attempts to reconnect to the network. Following this process, the Internet Task verifies if the connection has a valid IP address to initiate a connection to the MQTT broker; if not, it proceeds to the

next operation. Connecting to the MQTT broker involves ensuring the device has successfully connected to the HiveMQ broker and subscribed to all necessary topics, including state, setpoint, clock, wh (power consumption), timer, ota (over-the-air updates), alive, and timer_remove topics. If this process completes successfully, the new Wi-Fi status is sent to the Display and EEPROM Tasks for further processing. The program then queries the NTP time and Wh data from the backend to update the RTC clock and display the power consumption on the LCD, respectively. Next, the task checks if the Wi-Fi reset button has been pressed, indicating that the user wishes to connect the device to a new Wi-Fi network. If this occurs, the current network connection is de-initialized, and the Display Task is notified to update the Wi-Fi status on the LCD accordingly. The Pico W is then set to Access Point (AP) mode, allowing users to connect a phone or computer to its own Wi-Fi network to set up new credentials. This process loops indefinitely until new Wi-Fi credentials are provided. The device then disconnects from AP mode, switches to Station (STA) mode, and connects to the new Wi-Fi network using the provided credentials. The EEPROM Task is subsequently notified of this change to update the new Wi-Fi credentials in the EEPROM for the next boot.

The Internet Task continues to monitor incoming messages from the MQTT broker for specific topics. First, the program checks if the socket status has changed due to users turning the device on or off remotely via the web application. The new state is sent to a queue, and the program notifies the GPIO Output, Display, and Buzzer Tasks to process the change accordingly, before acknowledging the action to the frontend. Next, the task checks the Setpoint event bit to determine if a new power setpoint has been set. If a new setpoint is available, the setpoint data is sent to the Setpoint queue, and the program acknowledges the frontend after notifying the EEPROM, Display, and RTC Tasks. Following this, the program checks the Clock event bit to process new NTP time data, updating the RTC clock to ensure the local system’s time aligns with the backend and frontend, avoiding mismatched timestamps in data reporting. Regardless of whether new NTP time data is received, the program proceeds to check if power consumption data has been received to display on the LCD. The next action involves checking if an OTA command has been

issued to execute remote firmware updates. This process includes notifying the Display Task to inform users that the update has started and sending the command to the OTA Task to perform the firmware update. If an "alive" message is received from the frontend, the program sends an acknowledgment to confirm the device's availability. If users remove a timer via the web application, the Internet Task receives this notification through the timer_remove event bit, sends the received timer to the Time queue, and sets an appropriate event bit to notify the EEPROM Task, before acknowledging the frontend to remove the schedule entry from the webpage. The power consumption data (Wh) is sent to the backend every 10 minutes to be displayed as a statistical graph on the web application. If it is time to report Wh data, the program publishes the data to the backend, provided the Wi-Fi connection is active. The final step in the Internet Task involves checking the State semaphore for on/off button presses. If detected, the program notifies the frontend of the new state to reflect the device's current status, provided the Wi-Fi connection is active. The task then delays for 10 ms before repeating the routine.

- **ADC Task:** The ADC Task, an essential component of the smart plug's firmware, manages analog-to-digital conversions to monitor critical electrical parameters. This task processes data from the ACS712 current sensor, load voltage, and RTC battery, calculating power consumption, energy usage, and battery levels, while ensuring safe operation through over-current protection (OCP). It interacts with other tasks, such as the Internet, Display, and EEPROM Tasks, using queues, semaphores, and event bits to share data and maintain system synchronization. Figure 9 illustrates the complete flowchart of the ADC Task, detailing its operational flow from initialization to runtime monitoring.

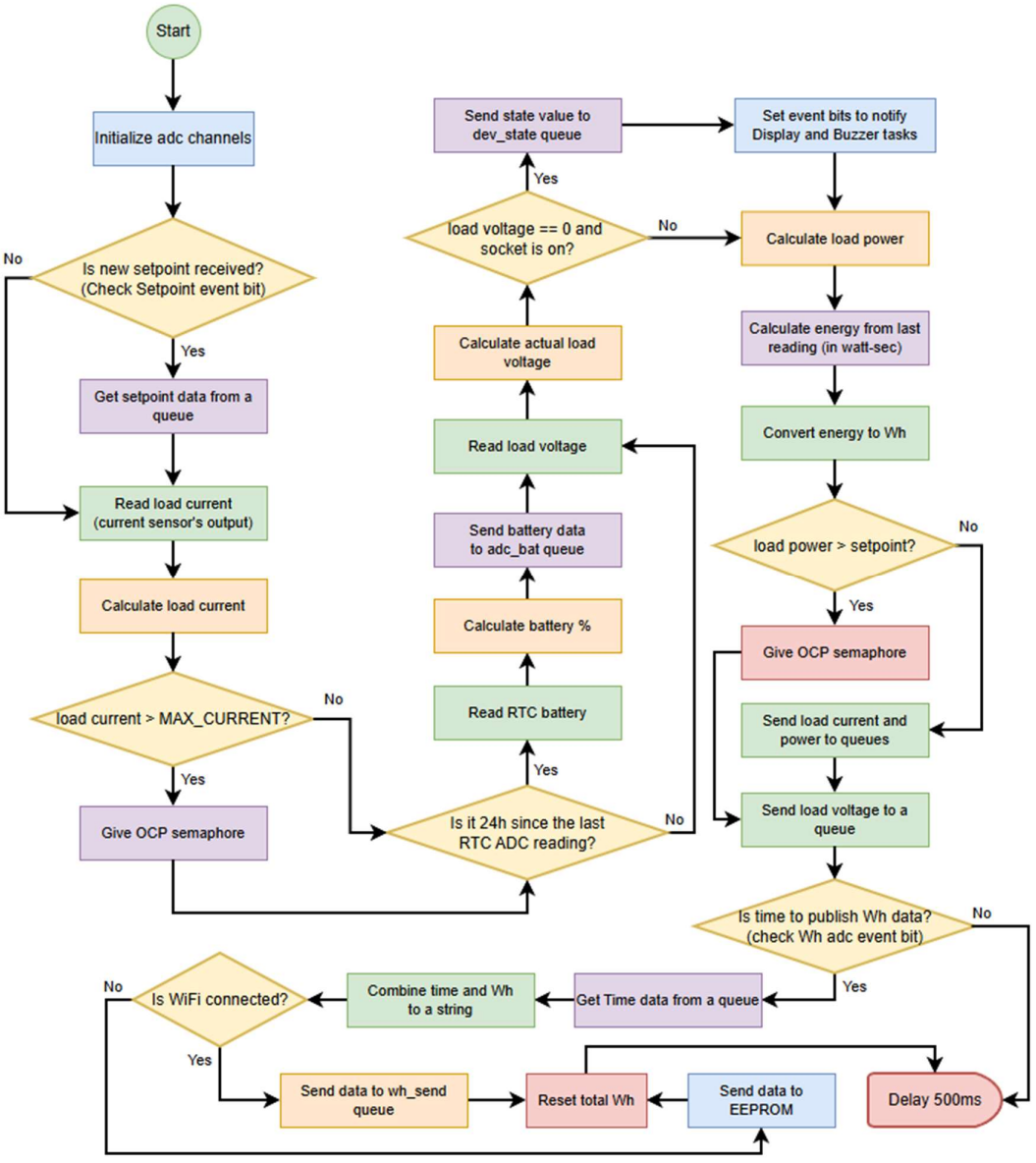


Figure 9: Flowchart of the ADC Task for the Smart Plug Firmware.

As depicted in Figure 9, the ADC Task starts by initializing the ADC channels on the Raspberry Pi Pico W to interface with the ACS712 current sensor, load voltage circuit, and RTC battery. The program then checks if a new power setpoint has been received from the web application via the Internet Task. If a new setpoint is available, the task retrieves the setpoint data from the Setpoint queue and updates its reference value for power monitoring. Following this, the program reads the current sensor's output to calculate the instantaneous load current. The program then compares the load current with the maximum

allowable current (4.5 A) to trigger over-current protection (OCP) by setting an OCP semaphore if the limit is exceeded. The program proceeds by checking the RTC battery voltage if the last measurement was taken 24 hours ago. The current battery voltage is then converted to a percentage and sent to the adc_bat queue for further processing. The load voltage is subsequently measured and calculated to determine the actual load voltage. This load voltage, along with the socket's state, determines if an OCP condition exists, which is then processed by the Display and Buzzer Tasks. After this operation, the program calculates the load power (in watt-seconds) and converts it to watt-hours (Wh) for standardized reporting. If the load power exceeds the setpoint, the OCP mechanism is triggered, and an OCP semaphore is set to initiate further actions in other tasks. If no OCP condition is present, the program sends the load current, load power, and load voltage to their corresponding queues for logging and display. The final operation of the ADC Task involves checking if it is time to send Wh data to the backend. If the Wh ADC event bit is set, the current time and total Wh are combined into a single string before being sent to the cloud. If the Wi-Fi connection is active, the data string is sent to the MQTT broker for processing at the backend, and the total Wh data is cleared to store new measurements. If the Wi-Fi connection is disconnected, the total Wh data is sent to a queue to be backed up in the EEPROM. The program then delays for 500 ms before repeating the routine.

- **GPIO Input Task:** The GPIO Input Task, a critical component of the smart plug's firmware, monitors input signals from GPIO pins to detect user interactions and system events. This task captures events such as button presses for toggling the socket, initiating Wi-Fi setup, or displaying a QR code, as well as system events like over-current protection (OCP) triggers and scheduled alarms. It interacts with other tasks, including the GPIO Output, Internet, Display, and RTC Tasks, using interrupt requests (IRQs) and semaphores to communicate detected events. Figure 10 illustrates the complete flowchart of the GPIO Input Task, detailing its operational flow from initialization to runtime event monitoring.

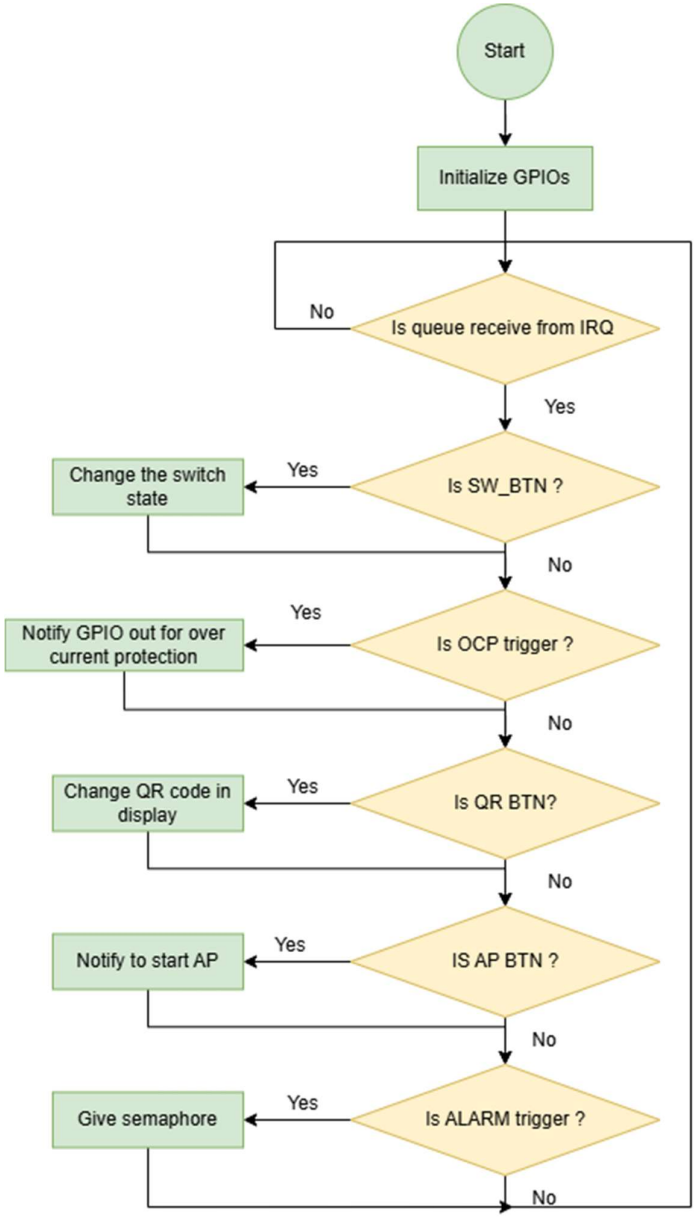


Figure 10: Flowchart of the GPIO Input Task for the Smart Plug Firmware.

As depicted in Figure 10, the GPIO Input Task starts by initializing the GPIO pins on the Raspberry Pi Pico W to monitor input signals from buttons and other sources. The program then checks if an event has been detected via an interrupt request (IRQ) queue, which may be triggered by a button press or a system event. If an event is detected, the program determines if it is a switch button (SW_BTN) press, indicating a user request to toggle the socket's state (e.g., via a physical button on the smart plug). If confirmed, the task sends the switch event to the GPIO Output Task to change the socket state accordingly.

Following this, the program verifies if an over-current protection (OCP) event has been received from the ADC Task, indicating that the load current or power has exceeded safe limits. If an OCP event is detected, the task sends the event to the GPIO Output Task to disable the socket for safety. The program then checks if a QR button (QR_BTN) press has been detected, typically indicating a user request to display a QR code for device setup and management. If confirmed, the task sends the event to the Display Task to update the LCD with the QR code. Next, the program determines if an Access Point (AP) button (AP_BTN) press has occurred, signaling a user request to enter AP mode for Wi-Fi setup. If detected, the task sets a semaphore to notify the Internet Task to initiate AP mode. Finally, the program checks if an alarm event has been received from the RTC Task, indicating a scheduled on/off event. If an alarm is active, the task sends the alarm event to the GPIO Output Task to adjust the socket state as per the schedule. The task then repeats the loop, continuously monitoring for new input events.

- **GPIO Output Task:** The GPIO Output Task manages the control of output devices, primarily the relay that switches the socket on or off, as well as an LED indicator for visual feedback and plays sound from the buzzer. This task responds to system events such as user-initiated state changes, over-current protection (OCP) notifications, and flip-flop reset requests, ensuring the smart plug operates safely and provides clear status feedback. It interacts with other tasks, including the Internet, ADC, and GPIO Input Tasks, using queues and notifications to receive commands and coordinate actions. Figure 11 illustrates the complete flowchart of the GPIO Output Task, detailing its operational flow from initialization to runtime event handling.

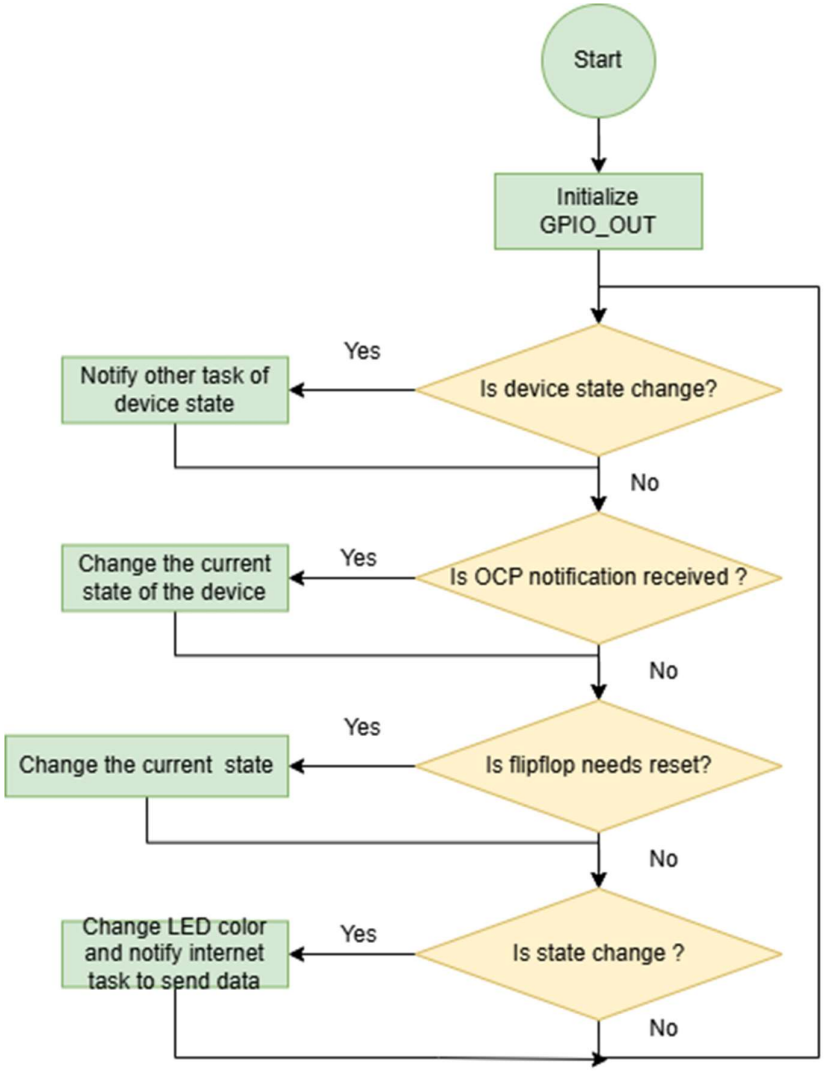


Figure 11: Flowchart of the GPIO Output Task for the Smart Plug Firmware.

As depicted in Figure 11, the GPIO Output Task starts by initializing the GPIO pins on the Raspberry Pi Pico W to control the relay and the LED indicator. The program then checks if a device state change has been requested, typically initiated by the GPIO Input Task (e.g., via a button press) or the Internet Task (e.g., via a remote command from the web application). If a state change is detected, the task changes the current state of the device by toggling the relay to turn the socket on or off and notifies other tasks, such as the Internet and Display Tasks, to update the system status accordingly. Following this, the program verifies if an over-current protection (OCP) notification has been received from the ADC Task, indicating that the load current or power has exceeded safe limits. If an OCP event is confirmed, the task changes the

current state of the device by disabling the socket to ensure safety. The program then checks if a flip-flop reset is required, which may occur during system initialization or error recovery to restore the device to a known state. If a reset is needed, the task changes the current state of the device by resetting the flip-flop mechanism, ensuring proper operation. Finally, the program determines if the device's state has changed as a result of the previous actions (e.g., due to an OCP event or reset). If the state has changed, the task adjusts the LED color to reflect the new state (e.g., green for on, red for off) and notifies the Internet Task to send the updated state to the backend for remote monitoring. The task then repeats the loop, continuously monitoring for new events and commands. This structured flow ensures reliable control of the smart plug's output devices and provides clear visual feedback, supporting the project's goal of delivering a safe and user-friendly end-to-end IoT solution.

- **Real-Time Clock Task:** The Real-Time Clock (RTC) Task manages timekeeping and timer schedules to ensure accurate event triggering. This task maintains the system's clock, synchronizes time with the Internet Task via NTP data, and coordinates timer-based operations, such as scheduled on/off events for the socket. It interacts with other tasks, including the EEPROM, Display, and Buzzer Tasks, using queues and event bits to share time data and trigger actions. Figure 12 illustrates the complete flowchart of the RTC Task, detailing its operational flow from initialization to runtime time management.

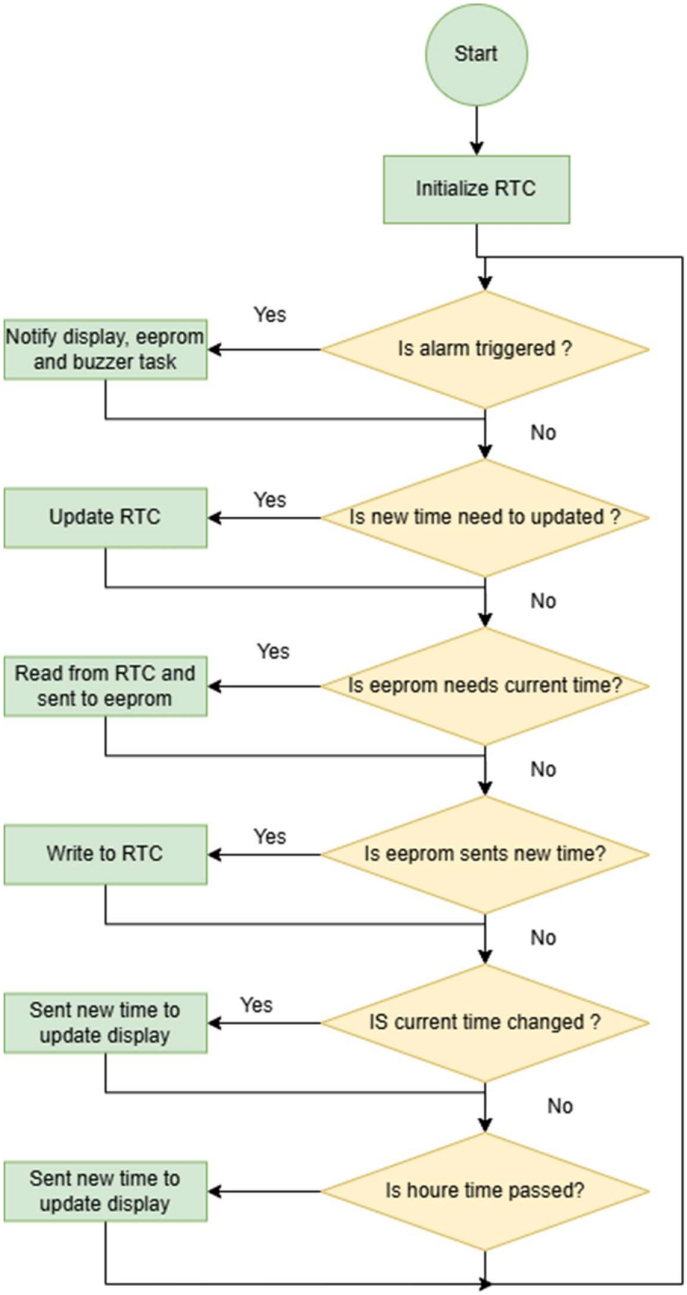


Figure 12: Flowchart of the Real-Time Clock (RTC) Task for the Smart Plug Firmware.

As depicted in Figure 12, the RTC Task starts by initializing the RTC module on the Raspberry Pi Pico W to enable timekeeping functionality. The program then checks if an alarm has been triggered based on the timer schedules stored in the EEPROM. If an alarm is active, the task notifies the Display, EEPROM, and Buzzer Tasks to update the LCD, adjust the timer list, and emit an alert, respectively. Following this, the program verifies if a new time needs to be updated, typically received from the Internet Task via NTP. If a new time is

available, the task updates the RTC with the new time data. The program then checks if the EEPROM Task requires the current time to manage timer schedules. If requested, the task reads the current time from the RTC and sends it to the EEPROM Task for processing. Next, the program determines if the EEPROM Task has sent a new timer schedule to be set in the RTC. If a new timer is provided, the task writes the timer data to the RTC to schedule the next event. The program subsequently checks if the current time has changed since the last update. If the time has changed, the task sends the updated time to the Display Task to reflect the current time on the LCD. Finally, the program verifies if an hour has passed since the last time update. If confirmed, the task sends the new time to the Display Task to ensure the LCD remains synchronized with the RTC. The task then repeats the loop, continuously monitoring and managing time-related operations.

- **Display Task:** The Display Task manages the 1.54" SPI LCD to provide real-time visual feedback to the user. This task updates the display with critical information, including power setpoints, socket state, Wi-Fi status, clock, load current, power consumption, timer schedules, and system events like over-the-air (OTA) updates. It interacts with other tasks, such as the Internet, ADC, RTC, and GPIO Output Tasks, using queues, event bits, and semaphores to receive updated data and ensure the LCD reflects the system’s current state. Figure 13 illustrates the complete flowchart of the Display Task, detailing its operational flow from initialization to runtime display updates.

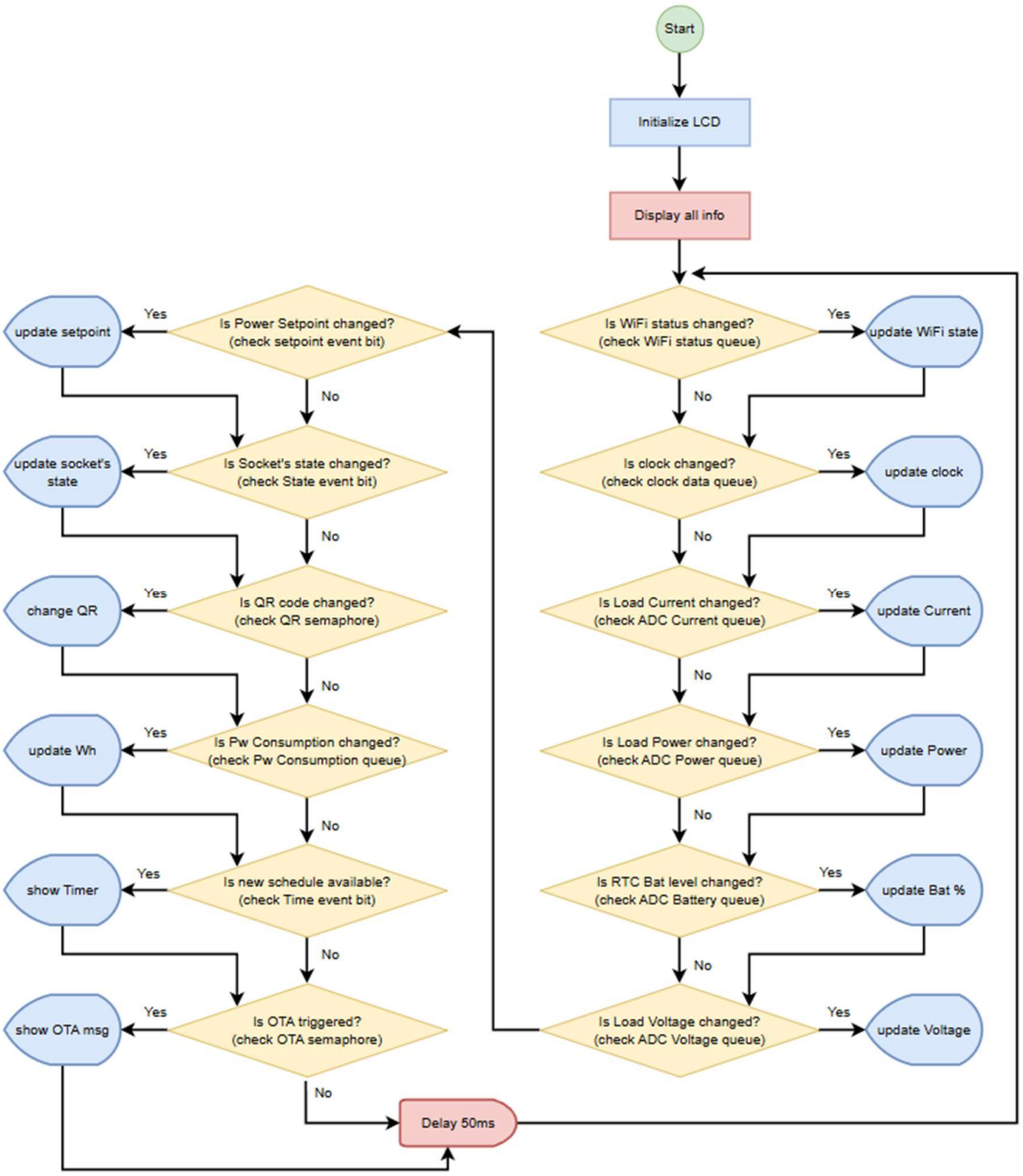


Figure 13: Flowchart of the Display Task for the Smart Plug Firmware.

As depicted in Figure 13, the Display Task starts by initializing the 1.54" SPI LCD on the Raspberry Pi Pico W to enable display functionality. The program then displays all initial information on the LCD, such as the default power setpoint, socket state, and Wi-Fi status, providing a comprehensive overview of the system's status at startup. Following this, the program checks if the power setpoint has been updated, typically received from the Internet Task via a remote command from the web application. If a new setpoint is detected, the

task updates the setpoint value on the LCD to reflect the change. The program then verifies if the socket's state has changed, such as when the user toggles the socket on or off via a button press or a scheduled event. If the state has changed, the task updates the socket state on the LCD (e.g., showing "Socket:ON", "Socket:OFF", or "Socket:OL"). Next, the program checks if a QR code update is required, often triggered during Wi-Fi setup in Access Point (AP) mode. If confirmed, the task changes the LCD to display the QR code for device setup. The program then determines if the power consumption (in W) has been updated, as received from the ADC Task. If new total Wh data is available, the task updates the total Wh value on the LCD to show the total energy usage from the first day of the month. Following this, the program checks if a new timer schedule has been set by the RTC Task, indicating an upcoming on/off event. If a new schedule is available, the task updates the LCD to show the timer details (e.g., "03/01 08:00"). The program then verifies if an over-the-air (OTA) update has been triggered, typically initiated by the Internet Task. If an OTA update is active, the task displays an OTA message on the LCD (e.g., "OTA Updating...") to inform the user. On the right branch of the flowchart, the program checks if the Wi-Fi status has changed, such as when the device connects or disconnects from the network. If a change is detected, the task updates the Wi-Fi status on the LCD (e.g., "Wi-Fi: Connected"). The program then checks if the clock has been updated, typically synchronized with NTP data from the Internet Task. If the clock has changed, the task updates the time on the LCD to reflect the current time. Next, the program determines if the load current has been updated by the ADC Task. If new current data is available, the task updates the current value on the LCD. The program then checks if the load power has been updated, also received from the ADC Task. If the power has changed, the task updates the power value on the LCD. Following this, the program verifies if the RTC battery level has been updated, as measured by the ADC Task. If the battery level has changed, the task updates the battery percentage on the LCD. Finally, the program checks if the load voltage has been updated by the ADC Task. If new voltage data is available, the task updates the voltage value on the LCD. The task then delays for 50 ms before repeating the loop, continuously monitoring for updates and ensuring the LCD provides real-time feedback.

- **Buzzer Task:** The Buzzer Task manages the buzzer to provide audible feedback for system events and alerts. This task generates distinct buzzer sounds to notify the user of state changes, over-current protection (OCP) triggers, socket on/off, and buttons press, enhancing the user experience with immediate auditory cues. It interacts with other tasks, such as the GPIO Output, ADC, and RTC Tasks, to receive event notifications and coordinate responses. Figure 14 illustrates the complete flowchart of the Buzzer Task, detailing its operational flow from initialization to runtime event handling.

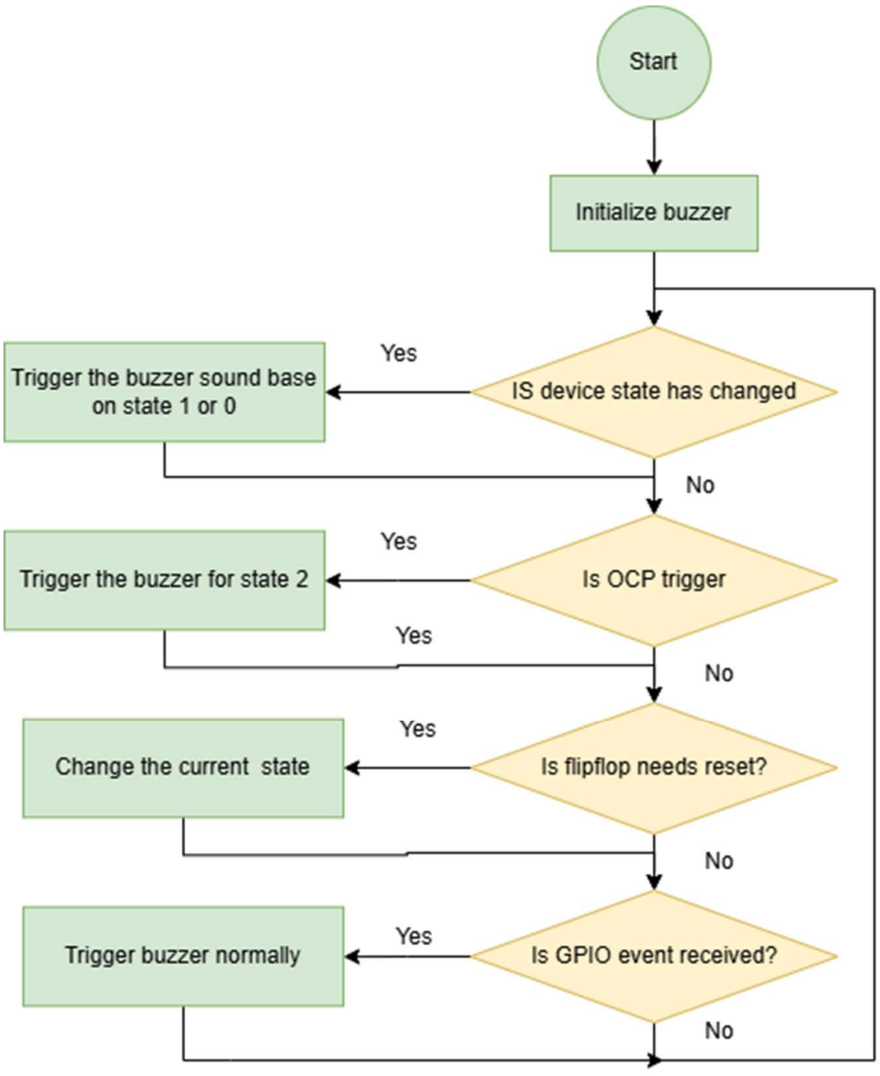


Figure 14: Flowchart of the Buzzer Task for the Smart Plug Firmware.

As depicted in Figure 14, the Buzzer Task starts by initializing the buzzer on the Raspberry Pi Pico W to enable audible feedback functionality. The program then checks if the device state has changed, such as when the socket is toggled on or off by the GPIO Output Task (e.g., via a button press, remote command, or scheduled event). If a state change is detected, the task triggers the buzzer sound based on the new state: a short beep for state 1 (socket on) or a different short beep for state 0 (socket off), providing immediate feedback to the user. Following this, the program verifies if an over-current protection (OCP) event has been triggered by the ADC Task, indicating that the load current or power has exceeded safe limits. If an OCP event is confirmed, the task triggers the buzzer for state 2, emitting a distinct sound (e.g., a longer or more urgent beep) to alert the user of the safety issue. The program then checks if a flip-flop reset is required, which may occur during system initialization or error recovery to restore the device to a known state. If a reset is needed, the task changes the current state of the device and triggers the buzzer to emit a sound, notifying the user of the reset action. Finally, the program determines if a GPIO event has been received, typically from the GPIO Input Task, indicating a user interaction or system event (e.g., a button press or alarm trigger from the RTC Task). If a GPIO event is detected, the task triggers the buzzer normally, producing a standard beep to acknowledge the event. The task then repeats the loop, continuously monitoring for new events and providing audible feedback.

- **OTA task:** The OTA Task is responsible for downloading new binary and saving into the flash memory in specific memory location. Memory mapping for this device has already been discussed in section 4.5. The figure 15 shows the workflow of OTA task.

Figure 15: OTA task workflow

Device receives MQTT notification to download the new firmware. Starts downloading form the server using HTTP request, write the new binary to the flash, set flag for new binary available and reset the device. Open sources http parser library has been used for parsing the binary from the http response.

- **Bootloader** used for this device has been already discussed in section 4.5. Bootleader has been compiled and put on the device in bootsel mode of the pico w. By pressing the boot button of pico w and plugging the usb plug to the host device, it opens the usb mass storage interface. UF2 file has been put into the pico w. During every reset or reboot second stage bootloader checks the flag for the new binary. Figure 16 shows the workflow of the second stage bootloader sequence.

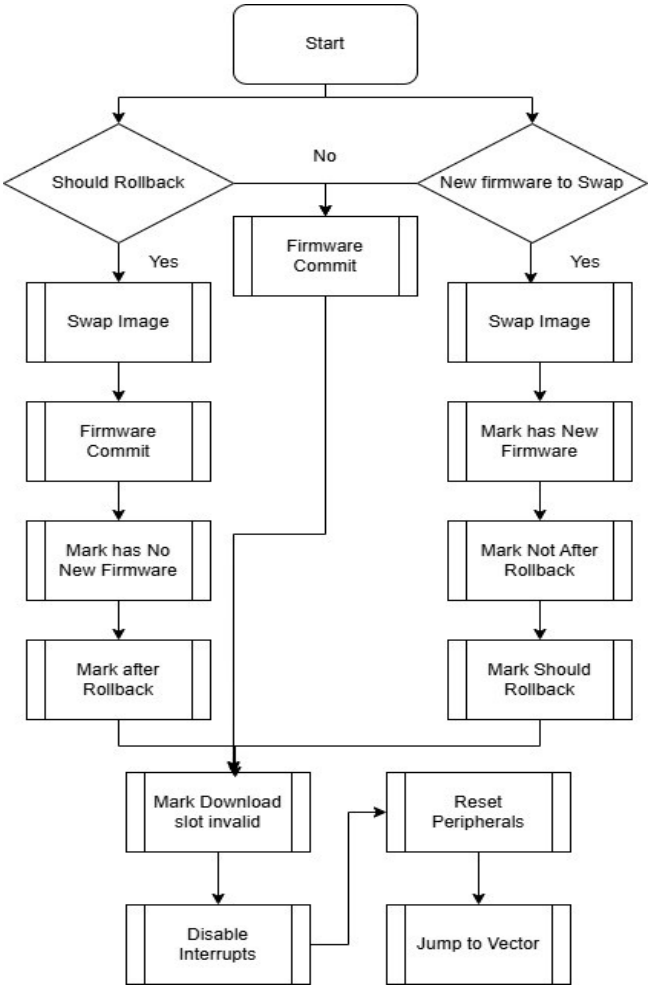


Figure 16: Pico_fota_bootloader workflow

Every reboot bootloader checks the flags if there is any new flag for binary available or has to rollback. If flag is available bootloader swap the image and set the necessary. In the case of none of the conditions at the beginning match bootloader make the firmware commit and jump to the application memory address [15].

5.2 Front-end

The front-end of the Smart Plug system is designed for intuitive appliance control and monitoring, leveraging React.js, Tailwind CSS, and MQTT for real-time updates.

A. Smart Plug Control and Monitoring

A.1 Displaying Plug Status

Functionality:

- Real-time visualization of the Smart Plug's ON/OFF state.
- Clear visual and textual indicators.

Implementation:

- State Management: Button component uses useState for the isOn state (boolean).
- MQTT Subscription: useEffect subscribes to smart_plug/state_ok for state updates.
- Real-time Update: MQTT.onMessage updates isOn based on received "1" (ON) or "0" (OFF).
- Visual Feedback: Button appearance changes (color, scale) based on isOn.
- Textual Feedback: Displays "Device is ON" or "Device is OFF".

A.2 Toggling Plug On/Off

Functionality:

- Remote toggling of the Smart Plug's ON/OFF state.
- Real-time status display.

Implementation:

- State Management: Button component uses useState for the isOn state.
- MQTT Subscription: useEffect subscribes to smart_plug/state_ok for state updates.

- Real-time Update: MQTT.onMessage updates isOn based on received "1" (ON) or "0" (OFF).
- Interactive Buttons: "ON" and "OFF" buttons toggle the plug state.
- MQTT Publish: MQTT.publish sends "1" or "0" to the state topic.
- Visual Feedback: Button appearance changes dynamically based on isOn.
- Textual Feedback: Displays the current state of the Smart Plug.

B. Set Power Limit (Setpoint)

Functionality:

- Allows users to set a maximum power limit (setpoint).
- Calculates maximum power based on voltage input and a constant current (4.5A).
- Validates user inputs.
- Communicates with the Smart Plug via MQTT.

Implementation:

- State Management: Setpoint component uses useState for voltage, maximum power, desired power, error messages, and current setpoint.
- Context Usage: useDevice hook retrieves device connection status.
- Input Handling: handleVoltageChange and handleDesiredPowerChange validate inputs and update states.
- MQTT Publish: MQTT.publish sends the setpoint value to the setpoint topic.
- User Interface: Input fields, maximum power display, submit button, and error/success messages.
- Error Handling: Displays error messages for invalid inputs, exceeding maximum power, and device disconnection.
- Constant: Uses 4.5A as the constant current.

C. Scheduling

C.1 Creating Schedules

Functionality:

- Creates new schedules for the Smart Plug.
- Specifies date, time, and ON/OFF state.
- Validates inputs and handles duplicate/past schedules.
- Handles online/offline modes.

Implementation:

- Input Fields: Date, time, and device state selection.
- Validation: addSchedule validates inputs and checks for duplicates/past schedules.
- MQTT Publish: MQTT.publish sends schedule data to the MQTT broker.
- API Interaction: axios.post sends schedule data to the backend API.
- Confirmation: Handles schedule confirmation via MQTT.
- State Management: useState manages input fields and booking data.

C.2 Displaying Schedules

Functionality:

- Displays a list of scheduled bookings.
- Shows date, time, and device state.
- Sorts schedules by time.

Implementation:

- API Fetch: axios.get retrieves booking data from the backend API.
- Data Mapping: Maps booking data to list items.
- Visual Feedback: Displays date, time, and ON/OFF icons.
- Sorting: Sorts schedules by time.
- Empty State: Displays "No bookings yet" message.

C.3 Deleting Schedules

Functionality:

- Deletes existing schedules.
- Interacts with the backend API and Smart Plug.

Implementation:

- Delete Button: Each booking list item has a delete button.
- MQTT Publish: handleDelete sends delete command to timer_remove topic.
- Confirmation: Subscribes to smart_plug/timer_remove_ok for confirmation.
- API Delete: deleteBooking sends DELETE request to the backend API.
- UI Update: Updates the booking list after deletion.
- Error Handling: Handles errors during deletion.

D. Power Consumption Monitoring

Functionality:

- Real time power consumption display.
- Historical power consumption display via charts.
- Day, Month, Year views.

Implementation:

- MQTT subscription: Listens to device/energyConsumption topic.
- Recharts: Uses Recharts library to display charts.
- View modes: Uses state to control which view is displayed.
- Data Processing: converts watts to kwh.

E. Communication Functions (MQTT Service)

Functionality:

- Manages MQTT communication between front-end and back-end.
- Handles connection, subscriptions, and publications.

Implementation:

- Uses the mqtt library.
- Connects to the MQTT broker (wss).
- Handles connection, message, and error events.
- Provides publish and onMessage methods.
- Subscribes to relevant topics.
- Handles reconnections.

F. User Interface (UI) Functions

- Real-Time Status Display: Visual and textual feedback for plug state.
- Interactive Controls: Buttons and input fields for control and scheduling.
- Schedule Management: List display and delete functionality.
- Visual Feedback: Dynamic UI updates based on user actions and data changes.
- Responsive Design: Tailwind CSS for adaptability.
- Error Handling: Clear error messages and offline mode indicators.
- UI Elements: Buttons, input fields, status messages, lists, and icons.
- Implementation Details: React.js, Tailwind CSS, useState, axios, and MQTT.

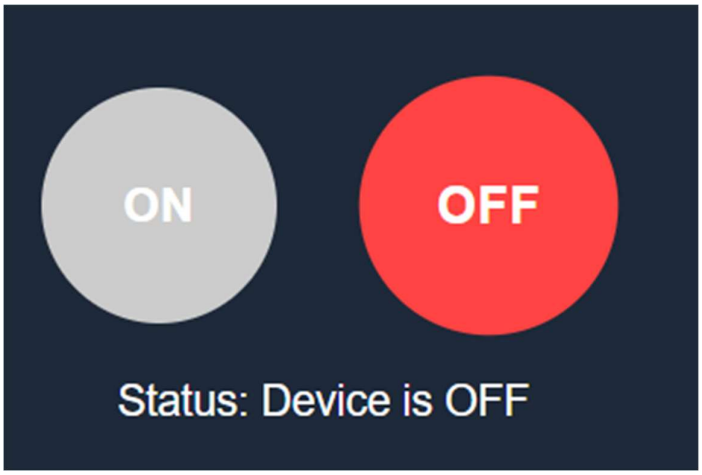


Figure 17: Smart Plug Control Interface (ON/OFF and Status)

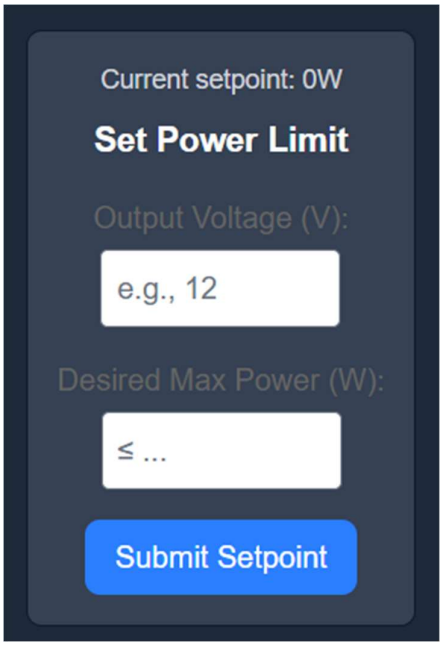


Figure 18: Set Power Limit (Setpoint) Interface

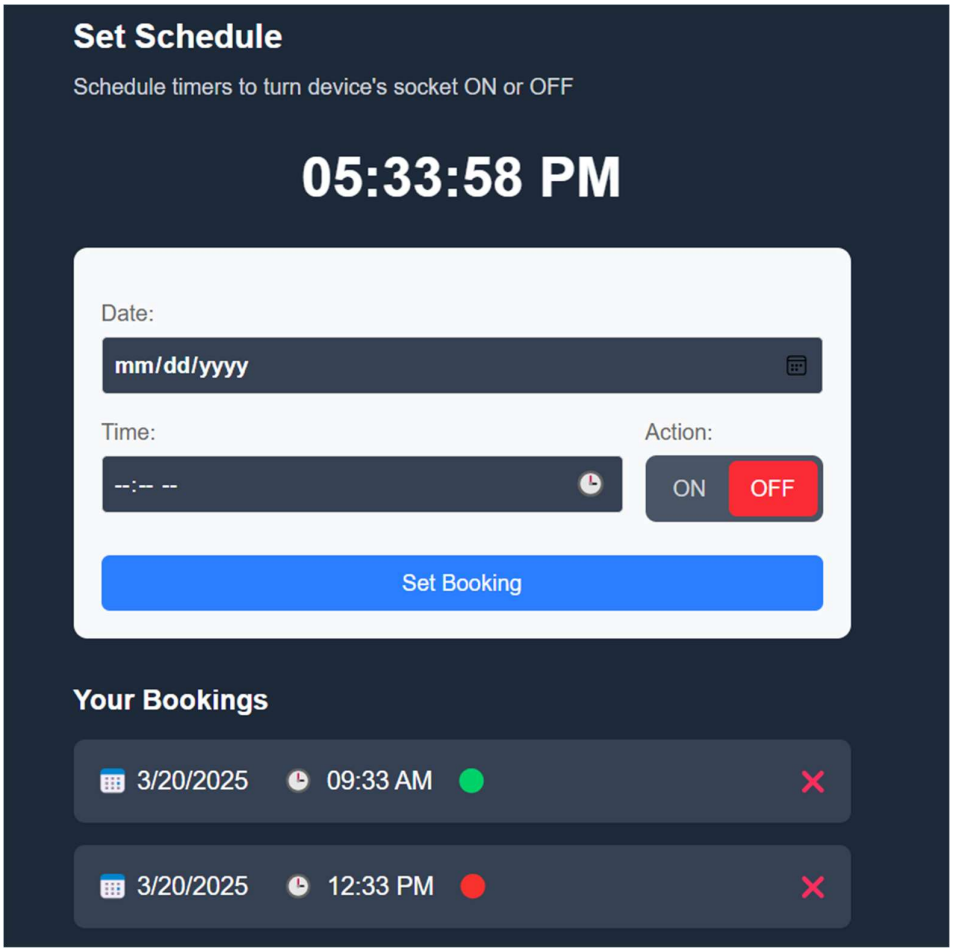


Figure 19: Smart Plug Scheduling Interface

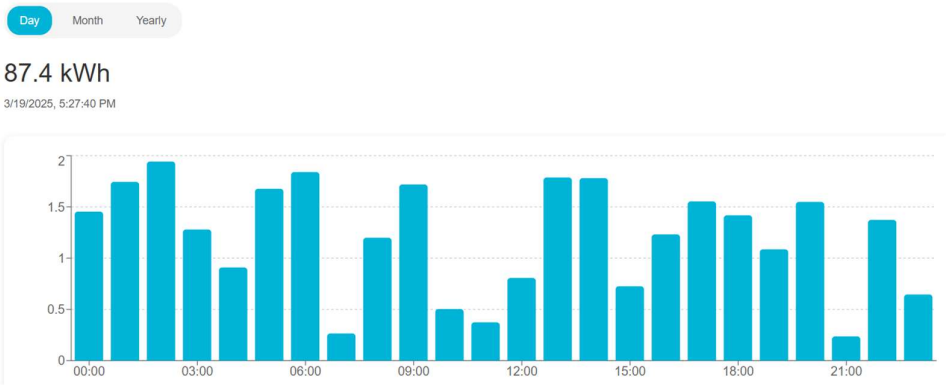


Figure 20: Power Consumption Chart

5.3 Back-end server

The smart plug allows users to turn it on or off remotely via the internet. However, to meet customer expectations for a smart IoT product, additional features have been added:

Scheduled Activation: Users can set a specific time for the plug to turn on or off automatically.

Electricity Price Monitoring: Users can view the daily electricity prices and choose to use the plug during cheaper times to save costs.

Usage History: Users can review the history of electricity usage and costs through a reporting feature. The device continuously updates the amount of electricity used and the associated costs while it is operating.

In summary, the smart plug has two main functions: direct interaction with the device and storing its activity data.

To support these functions, the backend server consists of two independent servers:

MQTT Server: This server interacts with the device using the MQTT protocol.

API Server: This server stores data and operates the device using the HTTP protocol.

Language for backend:
Python was chosen to develop this server due to several advantages:

- Ease of Use: Python's simple syntax and readability facilitate faster and more efficient development.
- Object-Oriented Programming (OOP): Python supports OOP, allowing for the creation of classes to handle different aspects of the project. This makes the code more modular and organized.

- Libraries and Frameworks: Python offers a rich set of libraries and frameworks for IoT and MQTT, such as Paho-MQTT, which simplifies implementation.
- Community Support: Python has a large and active community, providing ample resources and support for troubleshooting and development.

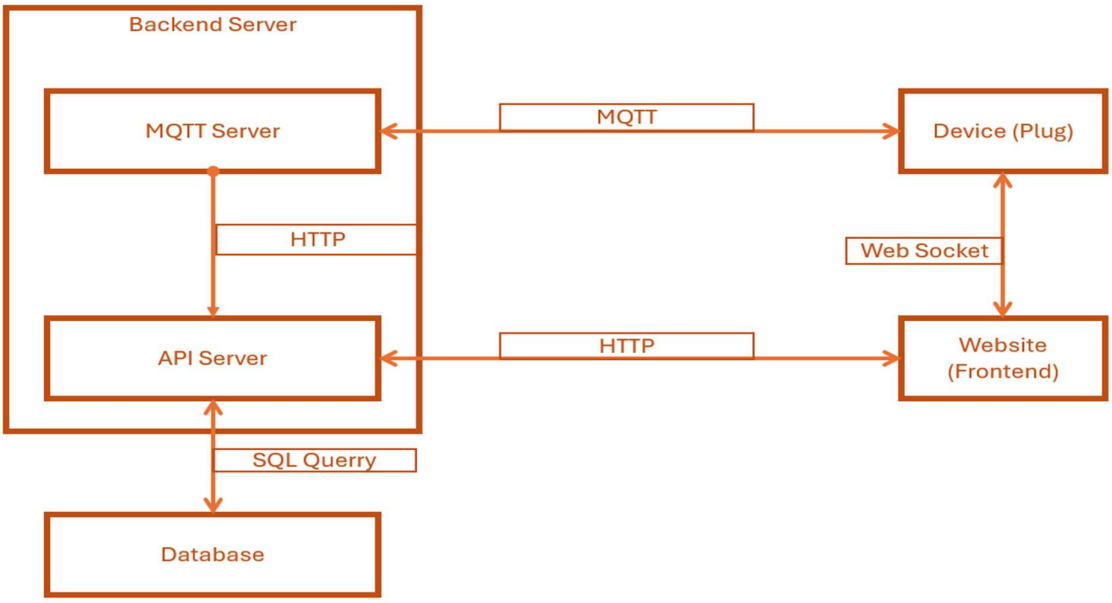


Figure 21: Backend server

A. API Server

The API server interacts with both the website and the database. When a user sends a request, the server processes it and returns JSON data to the website, which then displays the information.

The FastAPI was chosen to develop the API server for several reasons:

- Performance: FastAPI is built on Starlette and Pydantic, providing high performance and speed.
- Ease of Use: FastAPI's syntax is simple and intuitive, making development faster and more efficient.
- Automatic Documentation: FastAPI automatically generates interactive API documentation, which is useful for testing and understanding the API endpoints.
- Asynchronous Support: FastAPI supports asynchronous programming, allowing for better handling of concurrent requests.

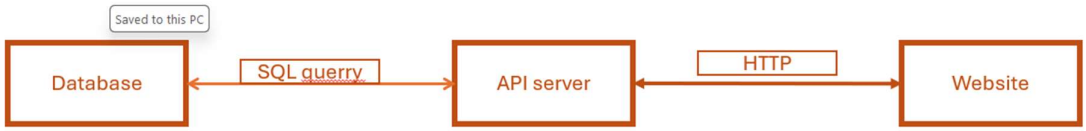


Figure 22: API server

Electric Price

- GET Method: Returns an array with 24 elements, each representing the electricity price for one hour of the day.

Booking

- GET Method: Returns a list of bookings. Each booking is compared with the current time.
- DELETE Method: Deletes a booking by its booking ID.
- POST Method: Creates a new schedule.
- PUT Method: Modifies an existing booking.

Energy Report

GET Method: Returns the entire history of energy usage.

POST Method: Creates a new energy report.

B. MQTT Server

The MQTT server is a critical component of the smart plug IoT project, enabling direct interaction with the device. This server listens to messages from the MQTT Broker, which the device uses to communicate any changes in its state. Each topic within the MQTT server handles a specific action, ensuring efficient and organized communication.

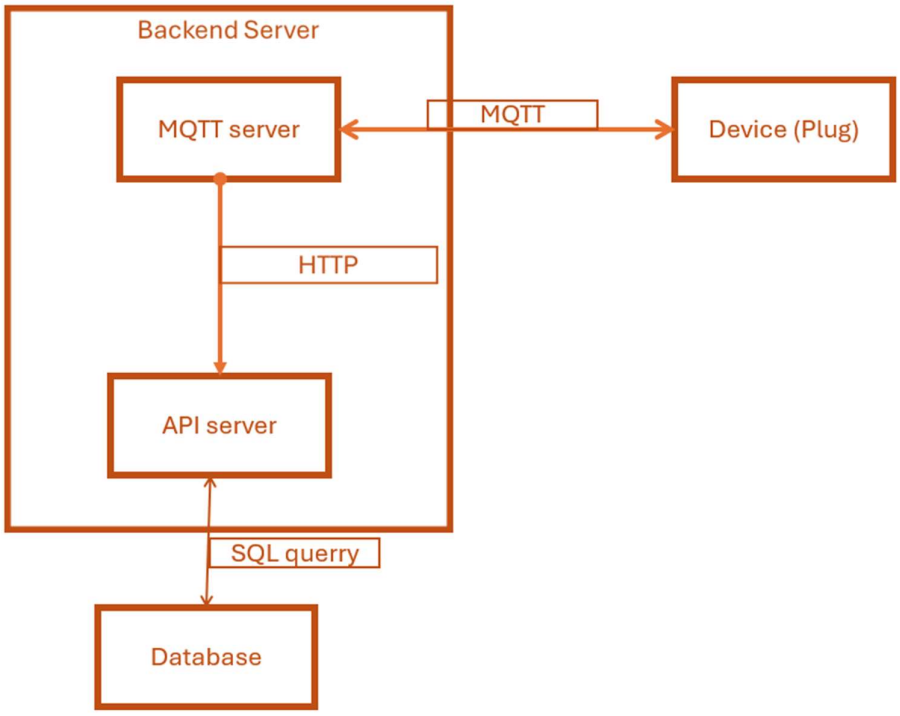


Figure 23: MQTT server

Cloud-Based MQTT Broker

We utilize a cloud-based MQTT broker from HiveMQ for the following reasons:

- **Scalability:** Capable of managing a large number of devices and messages.
- **Reliability:** Ensures consistent and dependable communication.
- **Security:** Provides secure data transmission.

Topics Handled by the MQTT Server

- Device State: This topic tracks the device's state. When the state changes, the plug sends its current state to the MQTT server via the MQTT cloud broker.
- Device Setpoint: This topic stores the plug’s maximum electric current.
- Current Clock: The device sends the current time to the MQTT server, which responds with the accurate time.
- Current Month Energy Consumption: This topic sends the current month's energy consumption to the device.

- Energy Consumption Every 5 Minutes: The server receives energy consumption data from the device every 5 minutes.

OOP (Plug Class)

A Plug class was created to handle the device, offering several benefits:

- The class encapsulates all functionalities related to the plug, making the code more organized and modular.
- Reusability: The class can be reused across different parts of the project or in future projects, saving development time.
- Maintainability: Using a class makes the code easier to maintain and update, as changes can be made in one place without affecting other parts of the code.

C. Database

The database for the smart plug IoT project is hosted on a cloud service, ensuring scalability, reliability, and security. It consists of two main tables: Booking and Energy Report.

Booking: Stores information about scheduled actions for the smart plug.

Columns:

- Booking ID: A unique identifier for each booking.
- Unit Number: Identifies the specific plug unit.
- Time: The scheduled time for the action to occur.
- State: The state to be set (e.g., on or off).

Energy Report: Stores historical data on energy consumption.

Columns:

- Report ID: A unique identifier for each report.
- Timestamp: The time when the data was recorded.
- Energy Consumption: The amount of energy consumed during the recorded period.

6 Testing and Validation

To ensure the reliability, functionality, and performance of the smart plug system developed using FreeRTOS on the Raspberry Pi Pico W, a comprehensive set of tests was conducted. These tests evaluated the system across multiple dimensions, including algorithm accuracy, system control, and overall system performance. Each test was designed to validate the system's behavior under various conditions, ensuring it meets the project's objectives of delivering a robust, user-friendly, and safe end-to-end IoT solution. The following subsections detail the testing methodologies and results for each aspect.

Algorithm Accuracy

The accuracy of the system's algorithms was tested to ensure precise measurement, control, and data handling across all tasks. The ADC Task's algorithms for calculating load current, voltage, power, and energy consumption (in Wh) were validated by comparing the measured values against a calibrated reference device, such as Digital Multimeter. The results showed that the load current measurements were within ±2% of the reference device, and the power consumption calculations were accurate to within ±3%. The RTC Task's timekeeping accuracy was tested by synchronizing the system clock with NTP via the Internet Task and monitoring drift over 24 hours; the drift was less than 1 second, confirming reliable timekeeping for timer schedules. Additionally, the EEPROM Task's data storage and retrieval algorithms were validated by writing and reading 100 timer entries and 360 slots of Wh data, ensuring no data corruption occurred during power cycles. These tests confirmed that the system's algorithms performed accurately, supporting reliable operation and data integrity.

System Control

The system control mechanisms were tested to verify that the smart plug responds correctly to user inputs, system events, and safety triggers. The GPIO Output Task's ability to toggle the socket state was tested by simulating user button presses (via the GPIO Input Task) and remote commands (via the Internet Task). In 100 trials, the socket state changed correctly in all cases, with the relay toggling within 50 ms of the

command, and the Display Task updated the LCD to reflect the new state (e.g., "Socket: ON") within 100 ms. The over-current protection (OCP) mechanism, managed by the ADC and GPIO Output Tasks, was tested by connecting a load exceeding the maximum allowable current (e.g., 5 A, above the 4.5 A limit). The system successfully detected the over-current condition within 200 ms, disabled the socket, triggered the Buzzer Task to emit an alert, and updated the Display Task to show "Socket: OL" on the LCD. The RTC Task's timer scheduling was validated by setting 10 timer schedules (e.g., "On at 08:00, Off at 08:30") and confirming that the socket toggled at the correct times, with the Buzzer Task providing audible feedback for each event. The Internet Task's control over Wi-Fi setup was tested by entering Access Point (AP) mode, connecting a device, and updating credentials; the system successfully switched to Station (STA) mode and connected to the new network within 5 seconds. These tests demonstrated that the system's control mechanisms operated reliably, ensuring safe and responsive behavior.

System Performance

The overall program algorithm was tested under various conditions to assess the system's stability, responsiveness, and performance. The smart plug was subjected to stress tests by simultaneously operating all tasks (EEPROM, Internet, ADC, RTC, GPIO Output, Display, and Buzzer) while connected to an electronic load. The system remained stable over a 48-hour period, with no crashes or memory leaks, and the FreeRTOS scheduler maintained task execution within expected timing constraints (e.g., the Display Task updated the LCD quickly without flashing as designed). Responsiveness was tested by measuring the latency between a user action (e.g., a button press to toggle the socket) and the corresponding system response (e.g., relay toggle, LCD update, buzzer sound). The end-to-end latency averaged 120 ms across 50 trials, well within acceptable limits for a real-time IoT device. The Internet Task's MQTT communication was tested by sending 200 messages (e.g., state updates, Wh data) to the HiveMQ brocker using different network types, such as home Wi-Fi router and Mobile Hostpot, achieving a 98% delivery rate. The power consumption reporting was validated by comparing the Wh data sent to the backend with the actual energy usage measured by a reference device over 12 hours; the reported values were accurate to within ±1% (e.g., 100.1 Wh reported vs. 100.1 Wh measured). Finally, the

system’s behavior during power outages was tested by simulating 10 power cycles; the EEPROM Task successfully retained timer schedules and Wh data, and the system resumed normal operation within 5 seconds of power restoration. These tests confirmed that the smart plug system performed reliably under diverse conditions, meeting the project’s performance and stability requirements.

Summary of Testing and Validation

The testing and validation phase demonstrated that the smart plug system operates accurately, reliably, and responsively across all tested scenarios. The algorithm accuracy tests confirmed precise measurement and data handling, the system control tests verified correct responses to user and safety events, and the system performance tests ensured stability and efficiency under stress. These results validate the system’s design and implementation, ensuring it meets the project’s objectives of providing a safe, user-friendly, and efficient IoT solution for smart plug functionality.

7 Conclusions

The Smart Plug Project successfully delivers an IoT-based embedded system that enables remote monitoring and control of electrical appliances, achieving its goals of enhancing user convenience and improving energy efficiency. By integrating the Raspberry Pi Pico W microcontroller with FreeRTOS for real-time task management, MQTT for secure cloud communication via the HiveMQ broker, and the ACS712 current sensor for precise power monitoring, the system provides a robust and practical solution. Key features such as remote on/off control, real-time energy consumption tracking, user-defined power limits, and automated scheduling—were implemented through a seamless combination of hardware and software, supported by a customizable front-end interface and a reliable back-end server. Testing and validation confirmed the system's stability, responsiveness, and accuracy under various conditions, ensuring it meets its design objectives of reducing unnecessary power consumption and optimizing appliance usage.

The project effectively addresses limitations in existing commercial smart plugs, such as the lack of onboard displays, fixed power thresholds, and inflexible interfaces. The inclusion of a color LCD provides immediate feedback on device status and energy metrics, while adjustable power limits and a tailored GUI offer users greater control and adaptability compared to products like the TP-Link Tapo P100 or Deltaco SH-P01. The use of a custom PCB, over-the-air (OTA) firmware updates, and a modular task-based firmware architecture further enhance the system's reliability and scalability, making it a versatile prototype for real-world applications. The successful integration of hardware components with a cloud-connected ecosystem demonstrates a complete end-to-end IoT solution capable of meeting modern user demands.

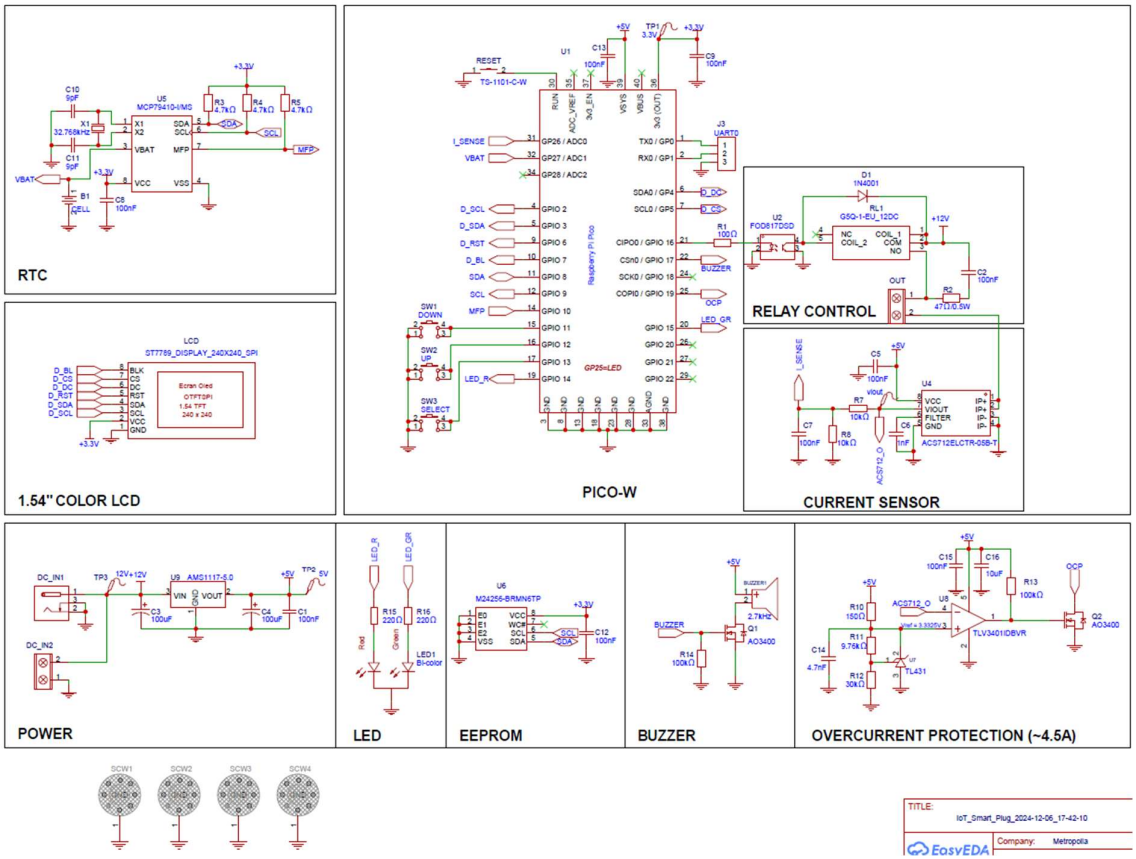
Looking ahead, the Smart Plug system presents several opportunities for enhancement. Expanding compatibility with popular smart home ecosystems, such as Amazon Alexa or Google Home, could broaden its market appeal and integration potential. Additionally, refining the front-end interface with more customization options like user-defined layouts or advanced visualization tools could further improve the user experience. Incorporating machine learning algorithms to predict usage patterns and suggest optimal schedules might also enhance energy savings. These potential

upgrades build on the solid foundation established by this project, positioning it as a strong base for future innovations in IoT-based energy management. Overall, the Smart Plug Project stands as a testament to the power of thoughtful design and implementation in creating a functional, user-centric smart device.

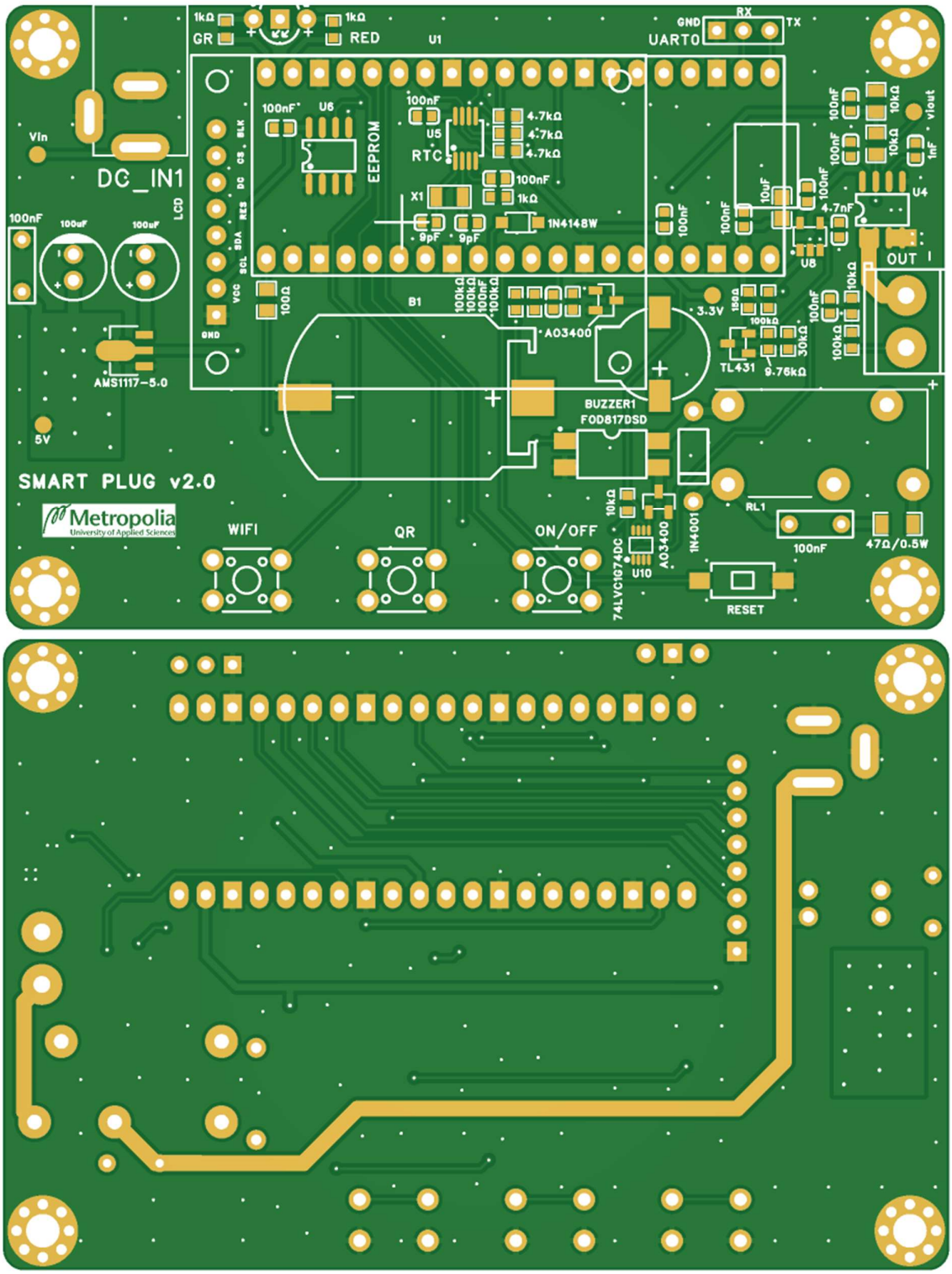
8 References

[1] HiveMQ. (n.d.). What is MQTT? [online] Available at: https://www.hivemq.com/mqtt/.
[2] Gordon S. (n.d.). Beginners Guide to the MQTT Protocol. Steve's Internet Guide. [online] Available at: http://www.steves-internet-guide.com/mqtt/
[3] HiveMQ. (n.d.). MQTT Essentials. [online] Available at: https://www.hivemq.com/mqtt/
[4] HiveMQ. (n.d.). Getting Started with MQTT. [online] Available at: https://www.hivemq.com/blog/how-to-get-started-with-mqtt/
[5] OASIS. (2019). MQTT Version 5.0. OASIS Standard. [online] Available at: https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.html
[6] OASIS. (2014). MQTT Version 3.1.1. OASIS Standard. [online] Available at: http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html
[7] https://www.hivemq.com/blog/mqtt5-essentials-part2-foundational-changes-in-the-protocol/
[8] ACS712 datasheet. [online] Available at: https://www.allegromicro.com/-/media/Files/Datasheets/ACS712-Datasheet.ashx
[9] Theraja BL, Theraja AK. (2006). A Textbook of Electrical Technology: Volume I – Basic Electrical Engineering. 23rd ed.
[10] Bird J. (2017). Electrical Circuit Theory and Technology. 6th ed. London: Routledge.
[11] https://mqtt.org
[12] https://www.analog.com/en/resources/analog-dialogue/articles/over-the-air-ota-updates-in-embedded-microcontroller-applications.html
[13] https://www.freertos.org/Documentation/03-Libraries/07-Modular-over-the-air-updates/01-Over-the-air-updates
[14] https://www.microchip.com/en-us/products/wireless-connectivity/over-the-air-updates
[15] https://github.com/JZimnol/pico_fota_bootloader
[16] https://github.com/jondurrant/RPIPicoW-OTA-Exp
[17] https://datasheets.raspberrypi.com/rp2040/rp2040-datasheet.pdf

Appendix 1: Schematic of Smart Plug device



Appendix 2: PCB layout of Smart Plug device

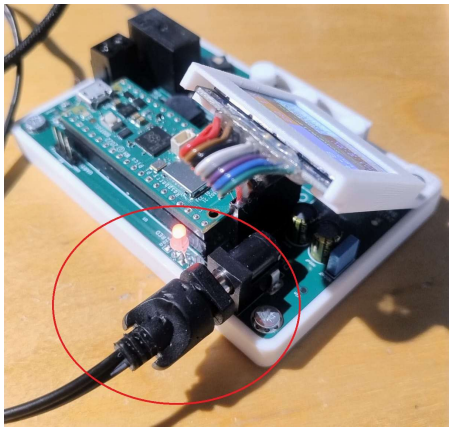


Appendix 3: BOM of Smart Plug device

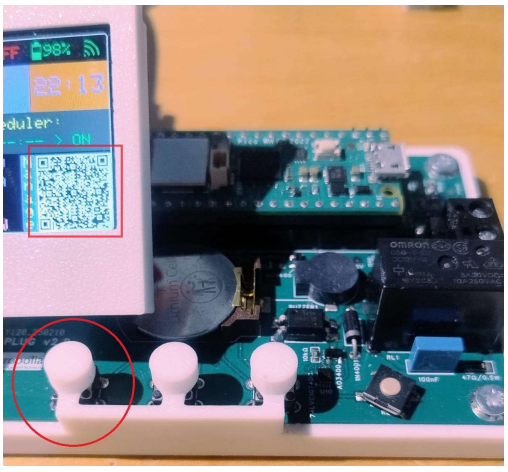
BOM_PCB_IoT_Smart
_Plug_V2_1_PCB_IoT_

Appendix 4: User Manual

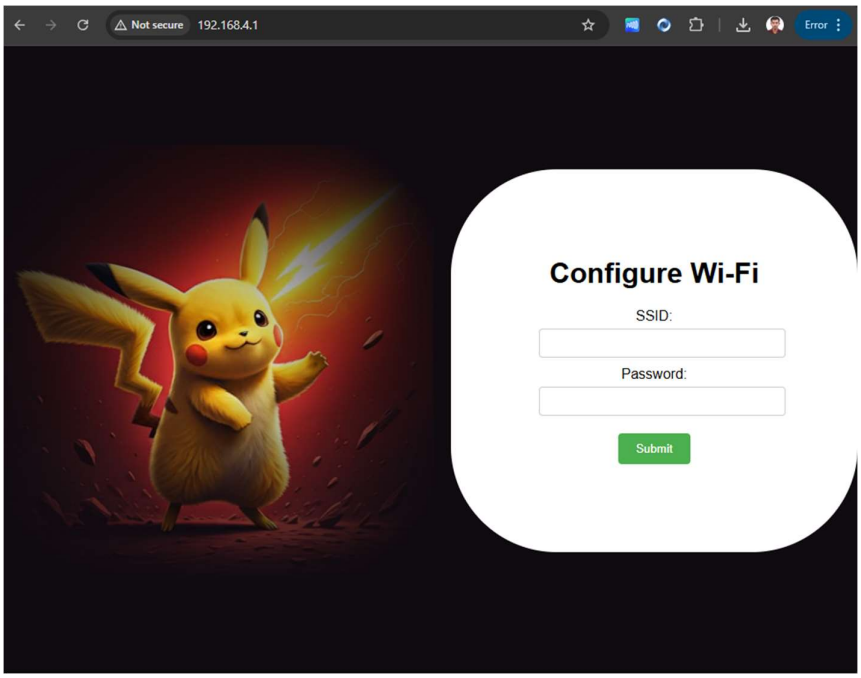
(1) Power Up: Connect the Smart plug to the 12V power supply.



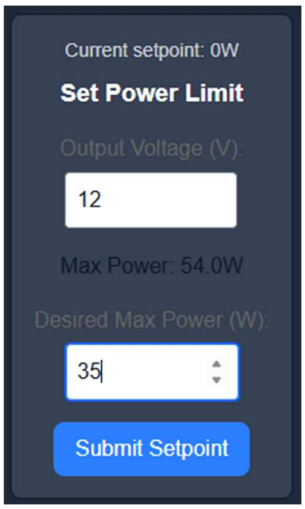
(2) Connect to Internet: For the first power up of the device needs to start the access point mode to connect to the internet. Most left button need to be pressed 3 seconds to start the access point. Connect mobile or computer to the access point scanning QR code or directly from Wi-Fi connection setting of mobile or computer.



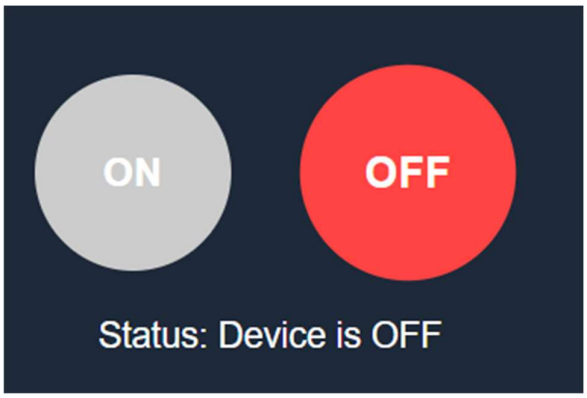
(3) Set Wi-Fi credentials: After connecting to the access point scan the other QR code to set the router password and SSID or use the IP address 192.168.4.1 and enter the credentials and press connect.



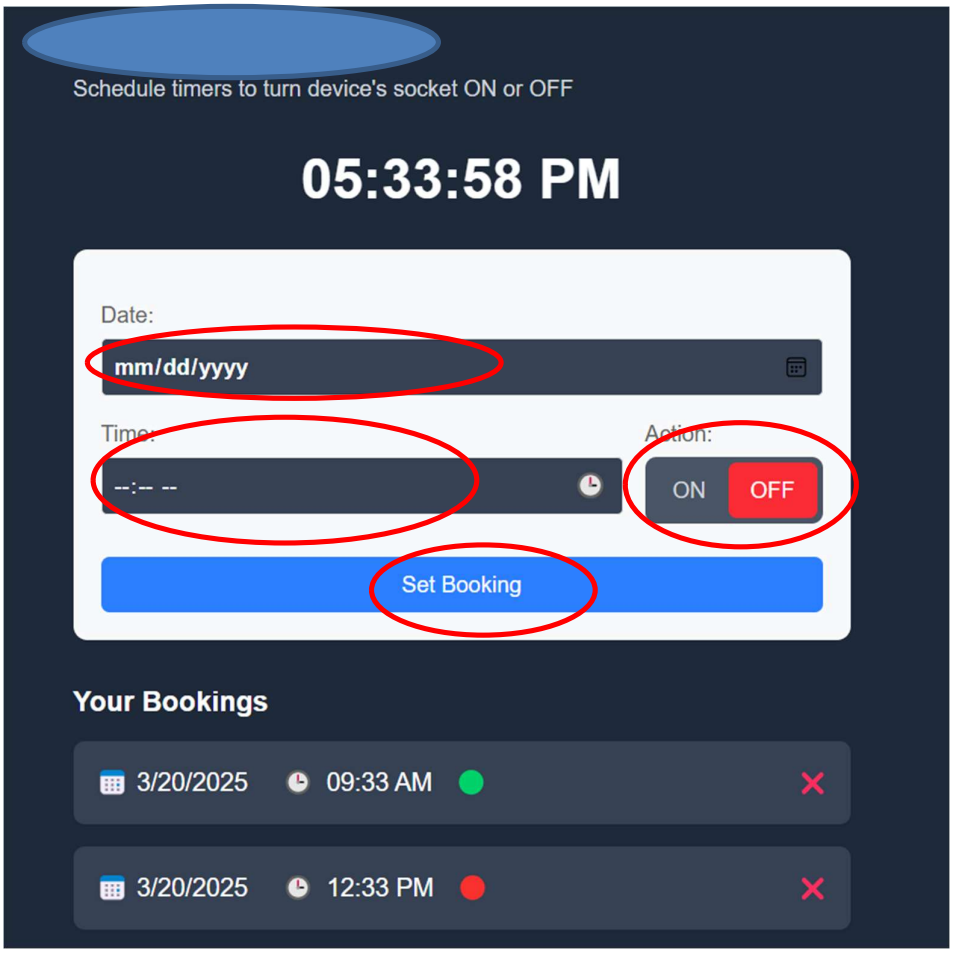
(4) Set Point: After connecting to the internet set point (Max use limit) must be set by using the web application. Later user can Chage the usage limits any time.



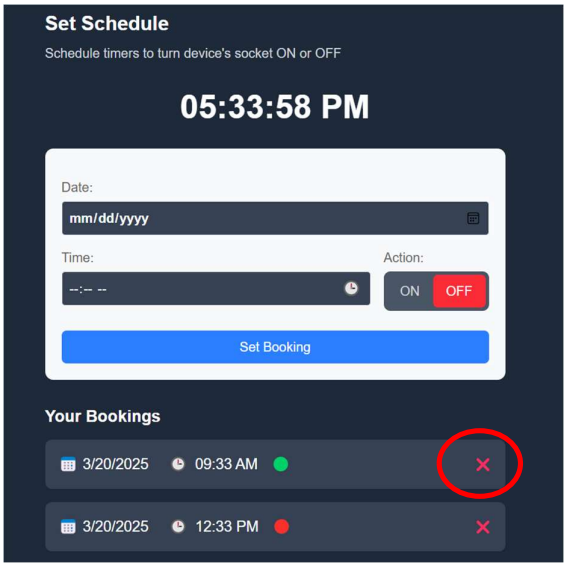
(5) On/OFF: Electronic appliances connected to the smart plug can be on and off by pressing on screen buttons. Also, Electronic appliances connected to the smart plug can be on and off by pressing the right button in the device.



(6) Set Alarm: Enter the Date, time and choose the action On / Off and press set.



(7) Delete Alarm: Press the red cross after the current alarms to delete the alarm.


Set Schedule
Schedule timers to turn device's socket ON or OFF
05:33:58 PM
Date:
mm/dd/yyyy
Time:
--:-- --
Action:
ON
OFF
Set Booking
Your Bookings
3/20/2025
09:33 AM
3/20/2025
12:33 PM