ADP - Charging Station Upgrade to OCPP Communication for Smart Charging

Report in the department of Mechanical Engineering by Junfan Jin, Can Zeng, Yunan Jiang & Huang Chen
Date of submission: May 11, 2025

1. Review: Prof. Dr. Stephan Rinderknecht
2. Review: M.Sc Thomas Franzelin
3. Review: Dr.Ing. Benjamin Blat-Belmonte
Darmstadt

TECHNISCHE UNIVERSITÄT DARMSTADT

Mechanical Engineering Department

IMS

LADESÄULEN-AUFRÜSTUNG AUF OCPP-KOMMUNIKATION FÜR INTELLIGENTES LADEN

TECHNISCHE UNIVERSITÄT DARMSTADT

ADP

für Yunan Jiang 2564706, Can Zeng 2387873, Junfan Jin 2629832, Huang Chen 2602451

Charging Station Upgrade to OCPP Communication for Smart Charging

---

Der Hochlauf der E-Mobilität bringt neue Herausforderungen für die Integration ins Stromnetz mit sich. Ladevorgänge können intelligent und netzdienlich gesteuert werden, sodass das Stromnetz entlastet wird. Als Demonstrator und Versuchsanlage soll dazu am IMS eine intelligente Ladestation entwickelt werden.

Eine bestehende Ladesäule soll daher im Rahmen dieser Arbeit in einen Demonstrator umgewandelt werden, an dem das Konzept und die Möglichkeiten von intelligentem Lademanagement von Elektrofahrzeugen gezeigt werden können. Des Weiteren soll es möglich sein, die Funktionsweise des Demonstrators flexibel anzupassen, um auf zukünftige Veränderungen der relevanten Standards reagieren zu können und diese im Rahmen der Forschung am IMS näher zu betrachten. Durch eine geeignete Möglichkeit der Messwertanzeige und Einflussnahme auf den Ladevorgang soll ein intuitives Anschauungsobjekt gestaltet werden, anhand welchem die Potenziale intelligenter Ladestrategien gezeigt und diskutiert werden können.

Aufgabenpakete:

- Literaturrecherche zu standardisierter Ladeinfrastruktur und intelligentem
- Bestandsaufnahme an der bestehenden Ladestation
- Konzipierung eines Ladesäulen-Upgrades für OCPP-Kommunikation
- Aufbau eines Demonstrators für intelligentes Lademanagement
- Test und Demonstration der Funktionsfähigkeit der intelligenten Ladestation

Bearbeitungszeitraum: 25.11.2024

Betreuung der Arbeit: Thomas Franzelin

Vergebene Credit-Points: 6

Darmstadt, 23.11.2024

Prof. Dr.-Ing. S. Rinderknecht

Contents

Abbreviations

CS Charging Station

CSMS Charging Station Management System

CP Charging Pilot

PP Proximity pilot

EV Electric Vehicle

GUI Graphical User Interfaces

OCPP Open Charge Point Protocol

V2G Vehicle-to-Grid

TX Transmission

RX Receiver

List of Figures

List of Tables

1 Introduction

1.1 Background

The development of internal combustion engine vehicles has spanned hundreds of years. However, with the advancement of Electric Vehicle (EV) technology and growing environmental awareness, more and more EVs are entering the consumer market.

By 2024, the sales of EVs had already reached six times those of 2018 [5], as shown in Figure 1.1. Similarly, the number of Charging Station (CS) increased by a factor of six, as illustrated in Figure 1.2.

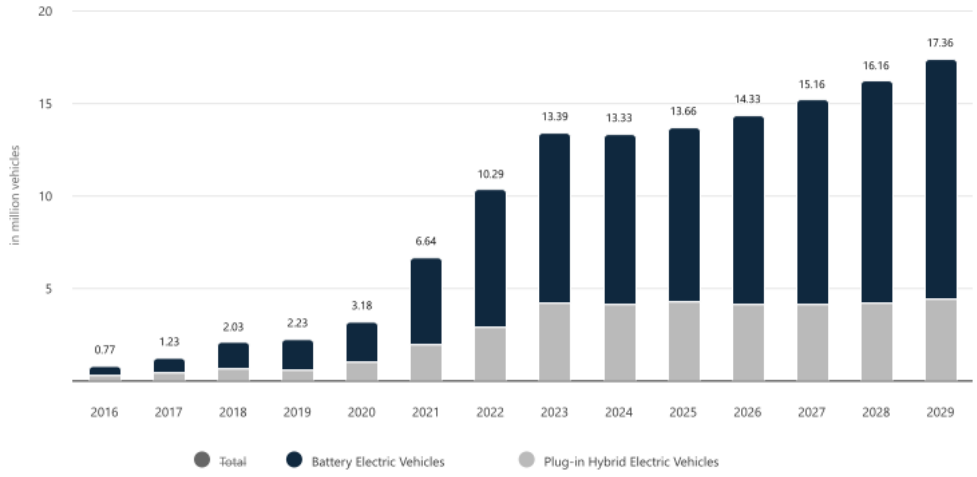


Figure 1.1: Electric vehicle sales increasing over the years (Source: [5])

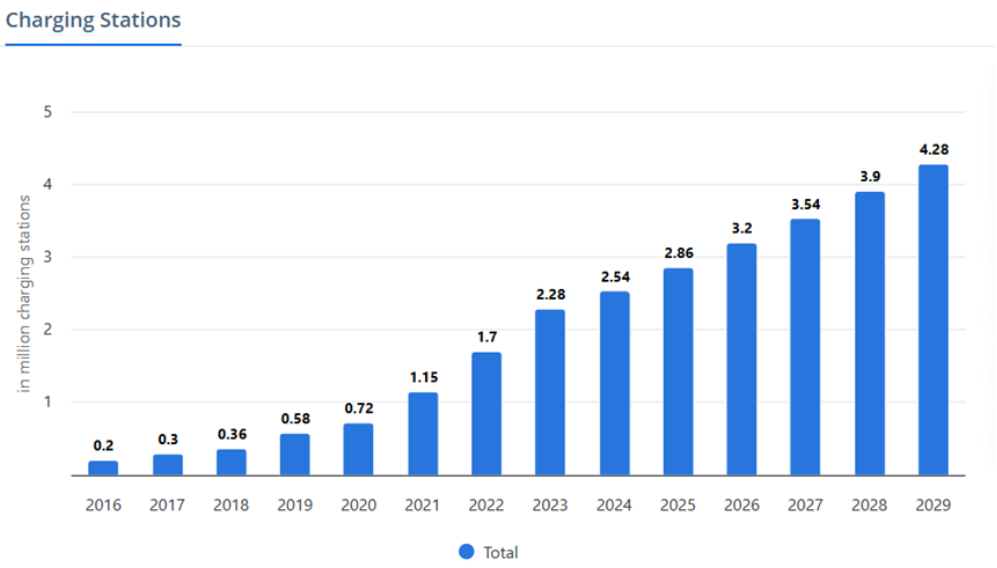


Figure 1.2: Charge station sales increasing over the years (Source: [5])

Nevertheless, the rapid growth in the number of EVs also poses significant challenges. Traditionally, charging adopts a "plug-and-charge" model, where the charging power remains constant. While the grid can accommodate charging demand at low EV penetration levels, the rising number of EVs is expected to impose a greater burden on the grid, particularly during peak demand periods.

In Germany, “dumb” charging of electric vehicles under a 10 million electric vehicles by 2035 scenario would lead to a 50% increase in low-voltage grid and transformer costs. (Schucht, 2017)[10].

To address these challenges, smart charging technology has emerged as a promising solution. Smart charging dynamically optimizes charging power by comprehensively considering multiple factors such as electricity price fluctuations, user preferences, charging demand, and grid conditions. By generating adaptive charging schedules for charging stations, smart charging helps to avoid grid load peaks and troughs, lower charging costs, and enhance grid stability.

In summary, smart charging is not only one of the key technologies for the large-scale development of electric mobility but also a crucial enabler for the future smart grid and the large-scale integration of renewable energy sources. Research on the modeling,

optimization, and control strategies of smart charging holds significant theoretical value and offers broad application prospects.

1.2 Research Significance

The necessity of smart charging lies in its ability to automatically generate a charging schedule based on known battery capacity and other relevant information. This enables the system to automatically adjust the charging rate according to different electricity pricing periods to realize peak shaving. This not only reduces charging costs but also reduces the peak load on the power grid. Large-scale regulation of the power grid requires control over all charging stations within a given region, which means that these stations must adopt a unified charging protocol. Therefore, it is essential to develop CS based on a standardized open-source protocol.

However, most charging stations do not use a unified communication protocol to standardize the charging process, which leads to compatibility issues when vehicles charge at different stations. Many manufacturers use their own closed-source charging protocols, such as the Nio Supercharger Protocol. The reliance on closed-source charging protocols significantly hampers compatibility and further limits the widespread adoption of smart charging.

The current mainstream open source charging protocol is Open Charge Point Protocol (OCPP) (refer to section 3.3), which is therefore applied in this project.

Under the OCPP framework, it is possible to dynamically adjust the output power of charging stations by sending control messages, thereby accommodating various charging strategies. Based on actual requirements, the charging duration and power can be flexibly adjusted, providing an operational foundation for the implementation of smart charging.

1.3 Objectives

The objective of this ADP is to modify an existing CS provided by the Institute for Mechatronic Systems (IMS) to achieve intelligent charging.

This CS is very primitive and can only be operated by two buttons: one for starting the charge and the other for stopping. The device could only charge at a fixed rate, and the charging process had to be started and stopped manually.

The main contents of this project are as follows:

- Search and read existing literature on standardized charging infrastructure and smart charging
- Learn about the existing CS
- Upgrade the old CS to support the OCPP communication protocol
- Complete the construction of a prototype of a smart charging system
- Test and debug the functions of the smart CS

After the charging station is upgraded, in addition to manual control for starting and stopping charging, it should also be capable of charging according to a preset charging schedule, dynamically adjusting the charging power, and even automatically generating an optimal charging plan based on user preferences or electricity prices, and executing it autonomously.

1.4 Overview

In Chapter 2, the hardware involved in the project and the overall connection structure of the CS are mainly introduced.

Chapter 3 introduces the basic architecture of the Charge Point System and the Optimizer System, as well as the communication methods between these two systems and the hardware. It covers how the OCPP Python library is used together with its limitations, how the system obtains OCPP messages and generates messages without every time to check JSON files through a generator, the implementation of signals, and asynchronous isolation.

Chapter 4 introduces some fundamental methods of the optimizer, the considerations involved in optimization, the definitions of charging cost, time, costs, constraints, and objectives, as well as the main function.

Chapter 5 mainly introduces the charging execution logic, module design, and functions, including the charging unit module, charging data manager module, thread management module for hardware polling, Modbus communication module, and GPIO manager module.

Chapter 6 mainly describes how to implement a Graphical User Interfaces (GUI) to display and manage charging data.

Chapter 7 introduces two steps to test the CS, namely simulation over software and testing in the lab, and the issues that occurred during testing and the results.

The last Chapter is about the evaluation of results, how many goals are achieved, and a discussion about the possible improvement of this system.

2 Hardware Introduction

The implementation of smart charging first requires hardware support. To achieve this, our smart charging station is built using the following hardware components: Raspberry Pi 3B (section 2.1), EVSE WB and Contactor AF30-30-00-13 (section 2.2), Shelly Pro 3EM (section 2.3), Power supplies of different voltages (section 2.4), level shifter (section 2.5), dual channel relay and DC motor (section 2.6), charge plug (section 2.7) and the original charging point.

These hardware components enable functions such as running embedded programs, controlling power output, supplying power to various modules, converting communication voltages, and measuring output power. We assembled these components on the track inside the charging station, and the simple connection is shown in Figure 2.1.

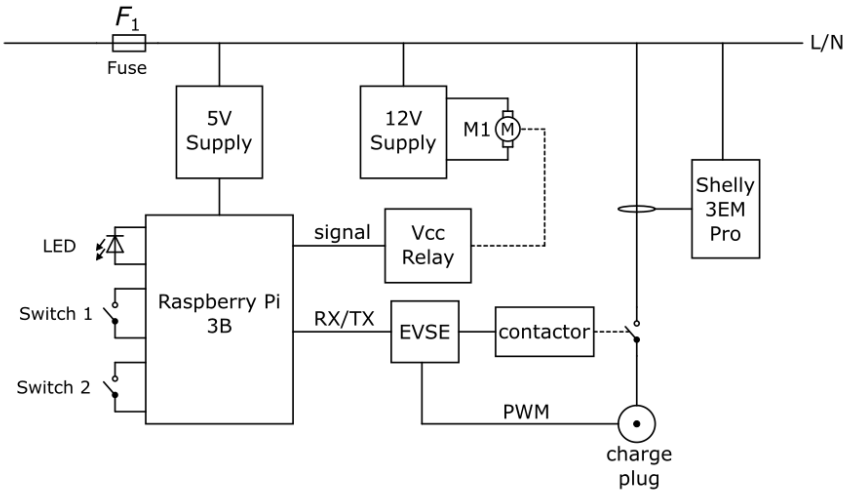


Figure 2.1: simple circuit

2.1 Raspberry Pi

Raspberry Pi is the carrier of the main charging program, which is used to connect, control, and communicate with other hardware (such as Shelly and EVSE), process or send the data obtained by the hardware (such as the total charged energy, current output, etc.), and process and report errors, etc. Raspberry Pi communicates with the Charging Station Management System (CSMS) through WebSocket and sends the target charging current value to EVSE according to the charging plan calculated by the optimization system. The Raspberry Pi model used in this project is Raspberry Pi 3B. The pin-out of Raspberry Pi 3B is shown in Figure 2.2.

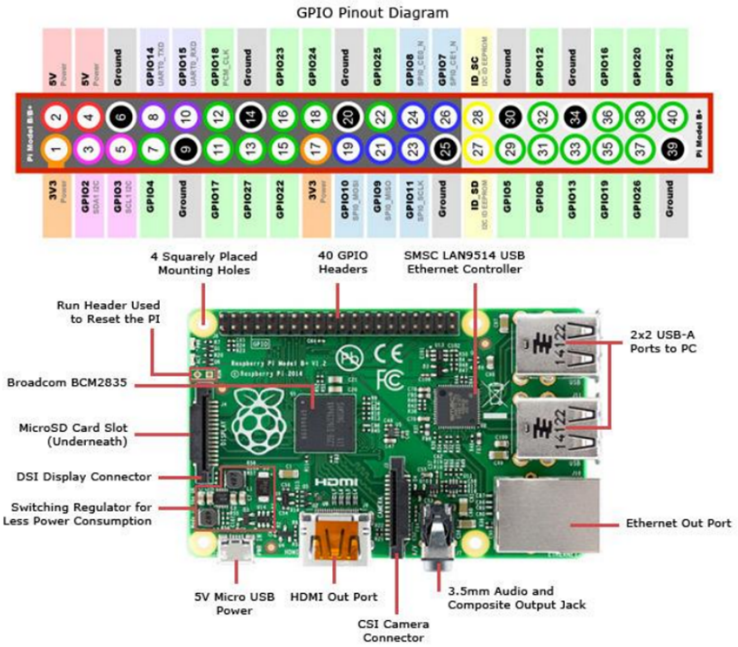


Figure 2.2: Raspberry Pi 3B and its pin-out overview (Source: [17])

Remote control of Raspberry Pi After the Raspberry Pi is loaded with the operating system, it is possible to configure the environment of the Raspberry Pi's Ubuntu system, install software, or perform other operations with an external monitor and keyboard. For easier debugging and commissioning, we use JupyterLab to access the Ubuntu system and

use it to launch the Python script. In order to achieve remote control, we first connect the Raspberry Pi with a fixed WiFi account and password. In this way, the Raspberry Pi can automatically connect to this pre-set WiFi after startup. And then another device (such as a laptop) can be connected to the same WiFi to remotely access the control interface of the Raspberry Pi by providing Jupyterlab the IP address of the Raspberry Pi to get access. By doing so, the Raspberry Pi becomes remotely controllable with only a power supply as a necessity. Using WiFi connection also facilitates communication between the Raspberry Pi and the Shelly sensor via HTTP requests.

The Raspberry Pi is connected to the EVSE via Modbus (Refer to section 5.3). By default, the primary UART of the Raspberry Pi 3B is assigned to the Linux console. To use the primary UART for other purposes, it is necessary to reconfigure Raspberry Pi OS by using raspi-config. After this configuration, we can use the serial port `/dev/ttyS0` and with the support of the pymodbus library to establish Modbus communication with the EVSE. [3] Due to the fact that the Raspberry Pi uses a 3.3V system and the EVSE uses a 5V system, these Modbus communication lines need to be connected with a level shifter to convert the voltage to avoid damage to the Raspberry Pi. Refer to the circuit diagram for the connection method (Ladestation_v2R).

Connection between Raspberry Pi and Shelly pro 3EM The Raspberry Pi and Shelly sensor are not physically connected in this project. Instead, the communication is achieved via HTTP requests. Since the charging station uses a three-phase AC power supply, we apply the Shelly Pro 3EM sensor, which provides three current clamps to measure real-time current and total charged energy for each phase. The Raspberry Pi obtains this data and monitors sensor status by sending periodic HTTP requests. This information is used to adjust the charging plan, detect errors, or display.

2.2 EVSE and Contactor

EVSE stands for electric vehicle supply equipment. It is an element that supplies electric energy for the recharging of electric or plug-in vehicles.

In order to adjust the current output, the hardware EVSE must be used to adapt the high-power input from the supply network to the user's side. In this case, EVSE WB is used which connects to the level shifter from 3.3V to 5V, obtaining the communication

signals with Receiver (RX) and Transmission (TX). RX and TX are directly controlled by Raspberry Pi.

EVSE transmits the allowed charging current value to the charging vehicle in the form of a PWM signal through the Charging Pilot (CP) terminal on the charging plug, and determines the vehicle status based on the feedback voltage level in this terminal. EVSE WB has 3 connectors. Connector X1 is connected to the high-voltage power grid to power the EVSE components (such as the relay that generates the PWM signal). Connector X2 includes the CP terminal, which leads to the same terminal on the charging plug, the Proximity pilot (PP) terminal for determining the allowed current, and other auxiliary terminals. There is also a PROG connector for control. In this project, we use the PROG connector for Modbus communication. Through Modbus communication, the Raspberry Pi can read and write the EVSE registers. This enables the Raspberry Pi to directly control the output current and obtain the vehicle and EVSE status. Since we will not change the cable of the charging plug, the maximum allowable current is limited to 13A (Although the cables can handle a current of 32A, the maximum current is limited to 13A to meet laboratory requirements.) by installing a fixed resistor between the PP terminal and the GND terminal of connector X2 [6].

AF30-30-00-13 (as shown in Figure 2.3) is connected to the rel pin on EVSE 4-pin side and controlled by the EVSE via the relay channel of X1. Its rated operating current is 50A AC[1], suitable for the requirements of this project. Unlike lower-powered relays, contactors typically have specialized structures for arc suppression, enabling them to interrupt heavy currents. With this contactor, the EVSE will only connect the circuit when the system is in a normal state.

2.3 Shelly Pro 3EM

Since the charging point uses three-phase AC, we chose Shelly Pro 3EM. As shown in Figure 2.4, Shelly Pro 3 EM can be assembled on DIN rail. It features three current clamps that measure the current and voltage of the three phases respectively, besides it also records the charged energy. Through HTTP requests, we can obtain measurement data as well as control the sensor. The data obtained from the Shelly sensor provides support for the correction and update of the charging schedule and is the basis of the dynamic charging plan adjustment of the smart charging point. After a charge is completed, we need to clear the accumulated energy consumption of each current clamp separately

Figure 2.3: AF30-30-00-13 (Source: [1])

through the `reset_totals` command and reset the total charged energy value to zero for the next charge.

Figure 2.4: Shelly Pro 3EM (Source: [14])

2.4 Power Supply (3.3V 5V 12V)

In this system, different hardware requires four kinds of voltage supply. The 220V network serves as the power source from which voltage levels such as 3.3V, 5V, and 12V are derived.

Firstly, STEP3-PS from Phoenix Contact provides 5V voltage for Raspberry Pi on the input side. Then, Raspberry Pi outputs 5V voltage as Vcc for the relay and the high-voltage side on the level shifter. Secondly, the Uno-power, a 12V DC power supply from Phoenix Contact, transforms 220V into 12V voltage which supplies the relay. Meanwhile, Raspberry Pi outputs signals with 3.3V on GPIO14 and GPIO15, which will be converted by the level shifter into 5V signals to EVSE. The electrical connection is demonstrated by the circuit diagram in the appendix.

2.5 Level shifter

The EVSE and Raspberry Pi communicate via Modbus, but because of the different operating voltages of the two systems, a level shifter is installed to convert the voltage between the Modbus communication lines to avoid damaging the Raspberry Pi. In order to facilitate fixed installation, we modeled and printed a mounting base to install the Raspberry Pi, the level shifter, and the relay together, as shown in Figure 2.5.

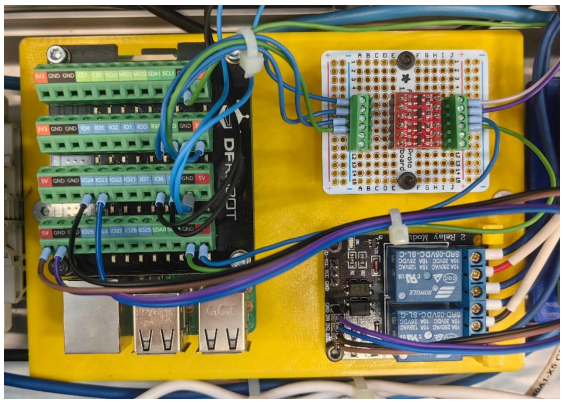


Figure 2.5: RaspberryPi board

2.6 Dual channel relay and DC motor

The DC motor is used to lock the charging plug to prevent the charging plug from being accidentally pulled out during the charging process. The DC motor is connected to a latch.

When the motor rotates forward, the latch is closed to lock the charging plug, and when the motor rotates reversely, the latch is released. The forward and reverse rotation of the motor is achieved by changing the direction of the voltage at both ends of the DC motor. For this purpose, we use a dual-channel relay to realize this function. The NO end of channel 1 and the NC end of channel 2 are connected to GND, the NC end of channel 1 and the NO end of channel 2 are connected to 12V voltage output, and the two ends of the DC motor are connected to the COM ports of the two channels, so that the direction of rotation of the motor can be changed by controlling the opening and closing of the relay. The forward and reverse rotation time of the motor is limited in the script to prevent the motor from being damaged by continuing to work for a long time after locking (or releasing). The relay is controlled by two GPIO terminals of the Raspberry Pi (GPIO23/GPIO24). The opening and closing can be controlled by simply pulling up the voltage level of the terminals (we use the LED command of the gpiozero library to complete this operation).

2.7 Connector

In this project, an IEC 62196-2 Typ 2 connector is used, which has seven pins. L1, L2, and L3 correspond to the three-phase power lines, while N represents the neutral line. PP and CP are signal lines for pre- and post-insertion signalling, respectively. PE is the ground wire.

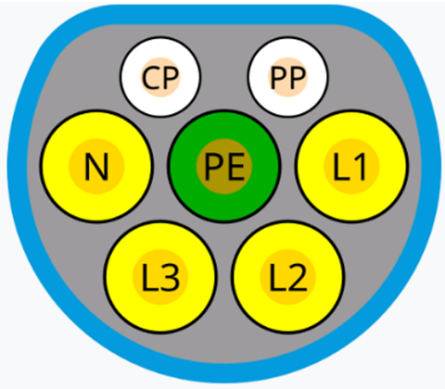


Figure 2.6: pin-out for typ2 connector (Source: [2])

There are other connector standards, such as SAE J1772 also known as IEC 62196-2 Typ 1, which is implemented in North America but only supports single-phase AC. Another example is Tesla's SAE J3400, which supports both single-phase AC and DC charging.

3 System Base

This chapter illustrates the division of the system structure using class diagram and how to use 4 modules from the official OCPP library. It gives an instruction to the workflow.

The workflow contains steps from obtaining OCPP data to generating OCPP data using message generator and how to adapt the signal-slot mechanism. Besides that, the synchronous and the asynchronous modules are isolated using message queue.

3.1 System Frame

This project features a relatively complex system architecture with numerous functionalities to implement, such as OCPP message processing and generation, charging logic control, and charging process optimization. Therefore, an effective solution is to decompose the entire large system into multiple subsystems, with each subsystem responsible for a major function[7]. This approach will significantly facilitate testing and debugging.

This project primarily consists of two core systems: the charging point system and the optimizer system. The functionalities and structures of these two systems are similar, as illustrated in Figure 3.1.

The Optimizer System is composed of two main modules: the Communication Layer and the Service Layer. The Communication Layer is responsible for receiving and parsing various types of messages, including OCPP messages, images, and text messages. Additionally, it manages and maintains communication with the webserver to ensure smooth data transmission to the client interface. The Service Layer, which is the core of the Optimizer System, handles the primary business logic, such as calculating charging schedules, load balancing, and data analysis.

The Charging Point System, in addition to the Communication Layer and Service Layer, incorporates an additional Hardware Layer. The primary role of this layer is to manage

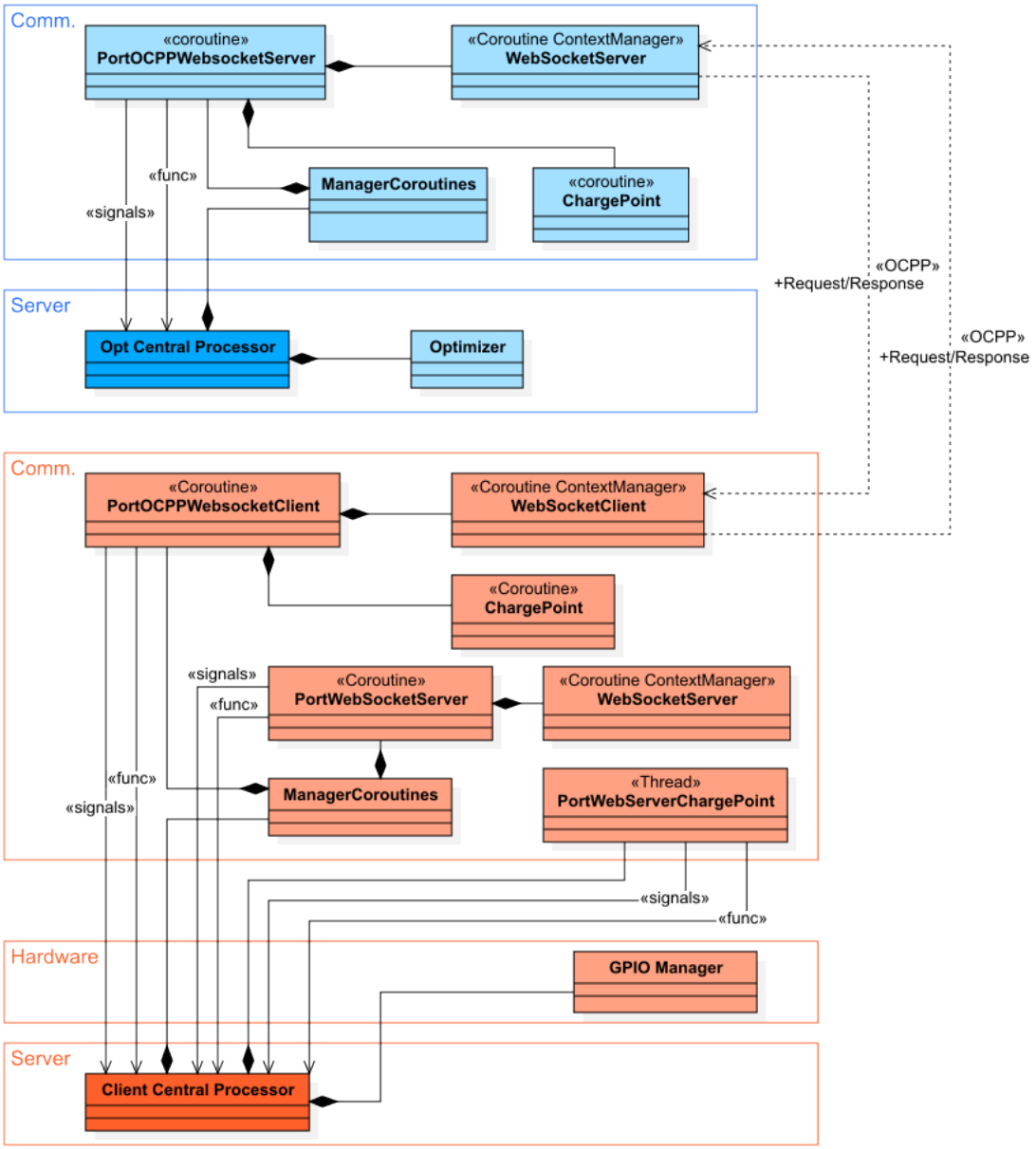


Figure 3.1: System Frame

hardware devices (such as charging point hardware and sensors) and facilitate communication with them. The integration of the Hardware Layer with other system components is crucial, ensuring that hardware devices function properly and provide real-time feedback. This layer also handles key operations such as switching the charging point on and off, executing charging schedules, and detecting faults.

In the Communication Layer of the Charging Point System, besides basic message reception and parsing capabilities, a `WebSocket` port has been specifically added. This `WebSocket` port is designed to support communication with new devices that may be developed in the future. This design enhances the system's scalability, ensuring that future additions of new devices or functionalities will not be restricted by the current communication method.

Additionally, the communication between the two systems is also realized via `WebSocket` port.

3.2 WebSocket

Thanks to full-duplex communication, `WebSocket` allows the client and server to send and receive data simultaneously without waiting for a response from the other side. Unlike HTTP's request-response model, once a `WebSocket` connection is established, data can be transmitted at any time, reducing communication delay[16]. The `WebSocket` connection remains open until either the client or server actively closes it.

However, instead of manually opening and closing the connection, the context manager is used in this project to automatically manage the connection. The opening and closing process of the `WebSocket` port is encapsulated in the `__enter__` and `__exit__` methods. Developers simply use the `with` statement; the connection is automatically established upon entering the context and closed upon exiting. And for the asynchronous case, `__aenter__` and `__aexit__` are used. And the resources are also released. This design not only simplifies the management process of the `WebSocket` port, avoids the problem of resource leakage when the connection closure is forgotten or an abnormal exit occurs but also enhances the readability and maintainability of the code.

3.3 Python Library OCPP

OCPP is a protocol developed and maintained by the Open Charge Alliance (OCA). In terms of device management, charging stations can send detailed diagnostic data to the CSMS, facilitating maintenance. It also utilizes ISO 15118 for secure vehicle authentication [9]. OCPP is used in this project due to its good compatibility, existing libraries in Python, and its open-source protocol. At the start of this project, Version 2.0.1 was the latest version of the protocol. Compared to OCPP 1.6, OCPP 2.0.1 offers more detailed message definitions in the JSON files, enhanced security features, and greater compatibility.

And the official OCPP library already exists in Python[11]. Though the latest official version of the library is v2.0.0, but before the start of this project, the latest library version was v1.0.0, that's why this project uses v1.0.0. Due to significant changes in the latest official library, it no longer supports the old version's enumeration classes. Therefore, the library used in this project is not compatible with the latest version.

The OCPP library includes 4 parts, namely the `ChargePoint` module, the `Routing` module, the enumeration class and data class of v1.6 and v2.0.1, and the error and information module. The procedure is to first register the method through the routing module, then start the `ChargePoint` module and obtain messages by continuously listening to `WebSocket`, or manually pass the information received by `WebSocket` to `ChargePoint`. The formal parameter of this registration method is the received message, and the returned value is the sent response message. The registration method is to decorate the registered function by using the decorators `@on(ocpp.enum.Action)` and `@after(ocpp.enum.Action)` defined in `ocpp.routing`. The decorator creates a registration method list in the corresponding `ocpp.enum.Action`, adds the decorated function name into it, and then binds the function to it when `ChargePoint` is instantiated. When the message is received and parsed, the bound function in the registration method list will be executed. The advantage of this method is that it is easy to use, requiring no additional registration settings or declarations. It is well-organized, and the message action to which the registration method is bound can be easily identified in the method definition.

However, the official library's function registration implementation method has several limitations:

1. no support for high-concurrency operations

In the code `ChargePoint._handle_call(self, msg)`, when the code tries to get the method registered with `@on`, it applies directly bound using a dictionary mapping. When the first message is being processed by the registered method and a second message with the same action is received, the second message has to wait for the first one to finish processing before it can be handled. As a result, when multiple clients send the same request, the system may experience a response timeout due to its inability to process subsequent messages in time.

However, modifying the OCPP library could introduce potential errors and increase the difficulty of porting the code to other devices, so we chose not to modify the library. Instead, we propose two possible solutions:

- Change the registration structure of the library without modifying the format of the message transmission. Register a factory when registering a method in `@on`. When a message is received, the factory should produce a method and then execute the produced method instead of the factory function. This separates and decouples the message input and execution. Then add a timestamp to the executed method, put it in the execution queue, and use another object to monitor the queue. If a timeout occurs, the execution of the method is forced to be interrupted, the method object is destroyed and removed from the queue, and a timeout signal is returned. When the list is empty, the listening will be stopped. In this way, by dynamically generating and managing execution objects, support for message handling when receiving multiple messages with the same action at the same time is achieved.
- Change the registration structure of the library and modify the format of the message transmission. Instead of using decorators to register methods, it is also possible to use signals (section 3.6) to connect them. Like the first approach, this separates the actions for handling and receiving messages. However, it requires the sender to send a timestamp with the data. Then add the execution method to the queue and start the listening loop. If a message times out, it is removed from the queue, and a timeout error is returned. The difference from the first method is that using signals can solve the problem of not being able to register multiple methods for one action at the same time.

2. no support for multi-function registration

   The code `ocpp.routing.create_route_map(obj)` uses a dictionary to define the mapping of the registration method. However, since the direct assignment

method `routes[action][option] = attr` is used, when a function is registered with the decorator of the same message action at the same time, for example `@on(Action.authorize)`, the function decorated later will overwrite the function decorated earlier. Therefore, it does not support registering multiple functions to handle the same message action.

3. all registered methods must be defined within the `ocpp.charge_point.ChargePoint` class

   The code `ocpp.routing.create_route_map(obj)` uses the `getattr(obj, attr_name)` method to obtain the registration method, and when instantiated in `ChargePoint`, the parameter passed to `routing.create_route_map(obj)` is `self`, that is, `ChargePoint` itself, so when the registration method is defined in other non-`ChargePoint` classes and their subclasses, it will not be registered, and the function will not be called for processing when receiving related message actions. Therefore, the registration method definition is strictly limited to the `ocpp.charge_point.ChargePoint` class.

3.4 Method to get OCPP data

In the OCPP official library, data is obtained by registering a method, where the first-level key name of the message is passed as a parameter to the registration method. These parameters are automatically unpacked and assigned when the message is received, providing the expanded data of one layer of JSON. To send a response, the registration method returns a data class containing the response data. However, this method couples the receiving, processing, and returning of messages, making it difficult to extend when the situation becomes more complex. To decouple this process, this project will use signals (see section 3.6) to separate the message reception. There are three types of OCPP message signals (Figure 3.2):

- `signal_request` is used to receive the request information sent by the requester. The specific content of the signal is a dictionary containing the timestamp, the message action, the message ID, and the message content.
- `signal_response` is used to receive the response information sent by the receiver. The specific content of the signal is a dictionary containing message action, request message, response message, timestamp of request message, and status of response message validity (successful response received, response timeout, reception error).

When the request sender times out while waiting for a response message, a signal data containing the response message validity status of the response timeout will be automatically received.

- `signal_response_result` is used to transmit the result after the receiver sends a response message. The specific content of the signal is a dictionary containing message action, timestamp of request message, response message, and sending result (successful response sent within the valid time, timeout sending, sending error). After a request message is sent, a response message will be returned. However, after the response message is sent, it is difficult for the system to know whether the response message is returned within the specified time. Therefore, based on the self-check in the system, the result of the response message will be fed back. This signal can be used to cache certain fixed data. When there are high-frequency requests, the data that was not successfully sent last time can be used directly to reduce the calculation time. However, this project does not have the need for high-frequency requests for the time being, so only the signal is created and not used.

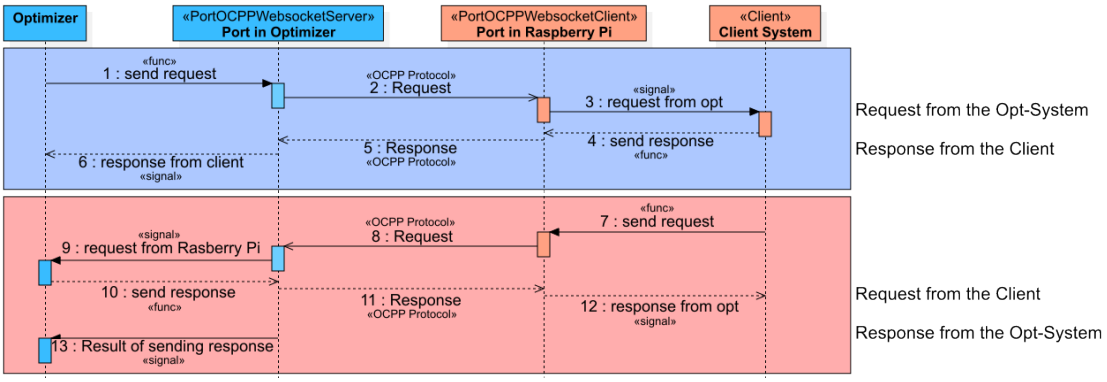


Figure 3.2: Message Communication

Since the original OCPP library code is not modified in this project, information is still obtained through registration. However, the return of the registration method is to call a `while` loop method, which loops to listen to the response message queue and checks whether a timeout occurs. When there is a message in the message queue, the program will exit the loop and return the message to the registration method. The registration method returns the response message and sends the response message through the `ChargePoint` class. The message queue is used to isolate synchronous and asynchronous programs, and

supports multiple messages waiting to be sent in the queue. The check time can set the estimated network delay time and the advanced sending time. For example, the standard timeout is 30s, the estimated network delay is 5s, and the advance sending time is 1s. The actual waiting time allowed is 24s. If the timeout exceeds, the default message will be automatically returned, and the timeout will be fed back to the system.

3.5 Method to generate OCPP data

Compared with the message format of OCPP 1.6, which has a simple hierarchical structure and only two layers at the maximum, the message format of OCPP 2.0.1 is significantly more complex in structure. The message hierarchy is deeper and not fixed, and sometimes exceeds four layers. Although the existing OCPP library provides corresponding message data classes for encapsulating message content, in the actual coding process, developers still need to manually assign values to these data classes or directly construct complex nested dictionary structures. This manual approach introduces significant risks and challenges. First, due to the extremely rich data fields in `OCPP 2.0.1`, there are a large number of type constraints, mandatory and optional restrictions between fields, and nested structure requirements. Developers are very likely to fill in the wrong fields, omit key parameters, or violate field constraints due to negligence during the manual construction process, which will cause OCPP messages to be rejected or communication fail during transmission. Secondly, when facing a large number of protocol standards, developers often need to frequently refer to the official JSON Schema or documents to confirm the field name, data type and its meaning, which increases development costs and reduces development efficiency.

In order to solve the above problems, reduce the error probability in the process of building OCPP messages, and improve development efficiency, this project designed and implemented an OCPP message generator that refers to the builder pattern[8] and fully utilizes the `DocString` prompt function of the IDE to create OCPP messages. The core idea of the generator is to encapsulate the complex structure of the OCPP message in the function interface. Developers only need to follow level-by-level prompts to fill in the data, automatically generating a data class or dictionary that conforms to the OCPP protocol standard, without worrying about the underlying nested structures or field constraints.

Specifically, the OCPP message generator exists in the form of a function. The formal parameters of the function are the field names of the single-layer data, and the actual parameters are the corresponding field values. The function automatically completes

the encapsulation and structure construction of the data according to the definition of the OCPP protocol, and finally returns a data object that meets the standard. Since the generator function is equipped with complete type annotations and document strings, when developers use mainstream IDEs to write code, the IDE can automatically identify the function's parameter type, restrictions, and field descriptions, and directly provide developers with complete prompts and verification during the code editing process, greatly reducing the development threshold and error probability.

Compared with the traditional way of manually building messages or consulting JSON Schema files, the OCPP message generator has the following advantages:

- Developers can quickly generate complete OCPP messages without repeatedly consulting protocol documents or JSON files.
- Type hints and IDE auto-completion functions can effectively avoid field spelling errors, data type errors, or structure mismatches.
- The generator function has a clear hierarchy and comments, and can be automatically regenerated when the protocol is updated later, improving the maintainability and scalability of the project.
- The generated code has a clear structure, is easy to read and understand, and reduces team communication costs.

It is worth mentioning that, considering the wide variety and complex structure of messages in the OCPP protocol, this project did not manually write generator functions one by one, but designed a set of automated scripts. This script can directly read the JSON Schema file officially released by OCPP, parse the field definition, nested relationship and data type of each message, and then automatically generate the corresponding Python message generator functions in batches. In the end, the project generated a total of 206 OCPP message generator functions, which fully covered all message types in the OCPP 2.0.1 protocol, greatly improving development efficiency and system integrity. For the newly released OCPP 2.1, although the OCPP library developers have refactored the enumeration class, only minor adjustments to the enumeration name adaptation rules in the generated script are required to support the new protocol version, eliminating the need for redevelopment from scratch.

3.6 Signal

The Signal and Slot was first introduced in the Qt framework to reduce coupling between different system modules. Qt is a cross-platform development toolkit, designed for creating both GUI and non-GUI applications [15]. In this project, we take advantage of the Signal and Slot mechanism, which allows components to interact without relying on each other. A single signal can be connected to multiple slot functions, while a single slot function can also be connected to multiple signals, making the system more flexible.

Compared to callback functions, signals allow dynamic connections and disconnections of slot functions, whereas callbacks must be defined beforehand, making it impossible to modify their calls during execution. However, signals connect to slot functions dynamically, it can be challenging to explicitly check signal connections, especially when the definitions of signal connections are distributed across different classes or modules. Thus, tracking calls of functions and methods becomes very difficult during the occurrence of bugs.

This project adapts the signal-slot mechanism from Qt, based on the observer pattern [8], to implement its own version. Since the system does not require a graphical interface and aims to minimize memory usage, the Qt signal-slot system is not used. The signals in this project are executed synchronously and do not support asynchronous execution. Unlike Qt's implementation, the signals here are not based on `QObject` or Qt's meta-object system, meaning the class containing the signal does not need to inherit from any specific class. As a result, while accessing data, especially in multi-threaded scenarios, there may be data safety concerns, leading to potential inconsistencies. However, given the system's small size and the limited shared data, this issue can be considered to have little impact. When a signal is activated within a thread, even if the slot function is defined in the main process, the slot will still execute within the thread. Unlike Qt's signal-slot system, this implementation does not require decorators for registration, supports both class-level and instance-level signals and signals do not need to be instantiated in `QObject`.

In this project, signals are implemented using two methods.

- The first implementation is a simple approach. In the signal class, a slot function list is created to store registered slot functions via the `connect()` method. At the same time, the slot function can also be unregistered in `disconnect()`. When the signal is triggered with `emit()`, the provided parameters are passed to all registered slot functions. These functions are executed sequentially in the order they were connected via `connect()`. If one of the slot functions blocks execution, the subsequent slot functions will also be blocked until the slot function stops blocking

and ends execution. The advantage of this method is its simplicity - it does not involve special Python mechanisms, it has a potential issue of overwriting assignment. To ensure safe usage, signals should be defined as private attributes within a class and exposed via `@property` methods. This prevents accidental overwrites but is limited to instance-level signals. Class-level signals still face the risk of being overwritten due to direct assignment.

- The second method utilizes Python's descriptor methods, `__get__` and `__set__`, to return the actual signal connection through the factory pattern. The factory pattern is a design pattern that creates objects through a factory class, which returns different types of objects based on input parameters, without the need for the client code to directly specify the concrete class [8]. Its advantage is that it only needs to be defined once, without the need to use private attributes and `@property` for protection. Additionally, the signal assignment can be disabled in `__set__`, which is still valid for class signals. Furthermore, all signals under the class can be managed through the factory. During debugging, all signals and connected slot functions can be viewed through the factory's signal list. However, this approach is more complex, requiring the use of special methods such as `__set__` and `__get__`. Since signals needs to be bound to the class, a list of all signal instances needs to be created in the class. This makes it not compatible with certain classes that strictly restrict class/instance attributes.

Both methods allow restricting the signal's parameter types during initialization. For example, a signal can be defined to accept only two parameters (`int|str, list`). If `emit()` is called with arguments that do not conform to the specified types, an error will be raised, minimizing the risk of unintended type-related errors.

The calling method of the OCPP library is introduced in section 3.3. This decorator method requires the execution methods to be defined in the `ocpp.ChargePoint` class, which means that ChargePoint not only needs to distribute the incoming messages, but also needs to execute the corresponding processing logic. This requires that the execution instance must be defined in ChargePoint, which means that the ChargePoint class is a top-level or upper-level class. However, in practice, there may be same-level cross-class calls or multi-process/thread execution, which will make the ChargePoint class very large and violate the single responsibility principle. For example, the optimizer calculation module can be implemented as a separate thread that passively waits for incoming data. This thread should exist independently rather than being bound to the lifecycle of ChargePoint.

Therefore, it is necessary to decouple the responsibilities of the ChargePoint class so that ChargePoint only unpacks and parses the message. The distribution of messages and

execution of command will be completed in the logical layer or higher-level classes. Under this design, WebSocket handles message transmission, ChargePoint processes message unpacking and analysis, and an upper-layer class manages message distribution, allowing for cross-class calls. This reduces the coupling between system modules and adheres to the Single Responsibility Principle. It also supports the possibility of extended development of multi-port and multi-process, and improves the expansion capability. Therefore, the signal-slot mechanism is adopted for internal communication within the system.

3.7 Asynchronous Isolation

A process can have multiple threads, and threads are used to execute multiple concurrent (assuming that the CPU has one core) or parallel (assuming that the CPU has multiple cores) operations within a single program. Different from threads, coroutines allow execution to be suspended and resumed. As shown in Table 3.1, while executing different tasks, coroutines don't need to switch the context or with a low switching-context cost. Context refers to the execution state of a program at a given moment, including register information, the call stack, memory data, and so on. When the CPU switches between different tasks (threads, processes, or coroutines), it needs to save the current task's state and restore the next task's state to ensure continuous execution [13].

Characteristic	**Multi-Thread**	**Coroutine**
Switching context cost	High	Low
Memory usage	High	Low
Execution Modes	Concurrency / Parallelism	Concurrency
Execution Layer	Kernel Mode	User Mode
Scheduler	Operating System Scheduler	Self-Scheduling by Programs

Table 3.1: Comparison between Thread and Coroutine

It is important to note that if a function uses the "await" syntax inside, it must be declared as "async" because "await" can only be used within a coroutine function. The higher-level function that calls this async function must also use await to wait for its result, which means the higher-level function must also become async. In other words, async and await always appear in pairs, and they can force functions that initially do not require asynchronous operations to be modified into asynchronous functions as well. However, most traditional programs are based on synchronous programming. The large-scale

introduction of asynchronous modes not only increases the difficulty of code management and maintenance but also reduces the scalability and readability of the program, causing the originally clear synchronous call chain to become complicated and difficult to trace. Therefore, it is very necessary to isolate asynchronous components from synchronous programs.

Since the `WebSocket` module is used in this project, and the `WebSocket` module itself is implemented based on an asynchronous mechanism, all method calls related are asynchronous functions (async methods). Consequently, modules that contain `WebSocket` interactions, such as the `ChargePoint` class, are classified as asynchronous programming modules. Other parts such as data processing, business logic, and control flow still use the synchronous programming mode to maintain the clarity and intuitiveness of traditional synchronous calls. In order to isolate these two programming modes, this project adopts the producer-consumer design pattern and transmits data through signals at the same time. The producer-consumer pattern decouples producers and consumers, allowing them to run independently and exchange data through a shared buffer. Producers do not know how consumers process data, and consumers do not know how producers generate data.

In this project, the producer is a synchronous module responsible for generating OCPP messages and processing data, while the consumer is an asynchronous module dedicated to transmitting and handling the generated OCPP messages. This pattern is primarily implemented in `sys_basis.Port_OCPP_WebSocket_Client.PortOCPPWebsocketClient`. The `PortOCPPWebsocketClient` class serves as the OCPP message client port and internally maintains a special message queue, `__list_request_message`, to store OCPP messages pending transmission. The core mechanism operates as follows:

When `start()` is executed, it starts an asynchronous coroutine task `__send_request_message`. This task continuously listens to the message queue `__list_request_message` within a `while` loop. When the message queue is empty, the `while` loop is blocked by the asynchronous event `__event_request_message`, entering a waiting state to avoid unnecessary polling and resource waste.

In the synchronous module, when an OCPP message needs to be sent, the synchronous method `send_request_message(message)` is called. This method places the generated messages into the message queue `__list_request_message` and releases the asynchronous event `__event_request_message`, which wakes up the `while` loop that is in a blocked state. After the asynchronous loop is awakened, it retrieves the messages from the message queue and calls the asynchronous method `self.__charge_point.send_request_message(self.`

`__list_request_message.pop(0))` to send the messages. When the queue is empty, the asynchronous event `__event_request_message` is set again to block the `while` loop.

Thus, the synchronous module and the asynchronous module are isolated through the message queue `__list_request_message`. The benefits of this approach are as follows:

- The synchronous module focuses on data generation and business logic processing, without concerning the asynchronous mechanism.
- The asynchronous module focuses on communication processing, specifically handling the asynchronous sending of messages.
- Both modules are decoupled through the message queue and event mechanism, avoiding the "contamination" caused by direct asynchronous calls.
- When the message queue is empty, the asynchronous loop is blocked again, until a new message arrives, ensuring the efficient use of system resources and reducing waiting cost.
- It improves the readability, maintainability, and scalability of the project code.

4 CSMS

In an electric vehicle charging system, the CSMS is the core software system responsible for managing and coordinating multiple charging stations, charging points, and user operations. It typically runs on cloud servers or local servers, providing charging operators with functionalities such as monitoring, control, billing, and maintenance[4].

CSMS serves as the "brain" of the entire charging infrastructure. However, in this project, the functionality of CSMS has been simplified, omitting authentication and data statistics while focusing on communication and optimization calculations.

The CSMS framework is divided into four main components: the main server function, OCPP communication, the optimizer, and the web interface. The OCPP communication and web interface are introduced in chapter 3 and chapter 6, respectively.

This chapter primarily introduces the key components of CSMS used in this project, including the utility class `DataGene`, the optimizer class `Optimizer`, and the main server function `Server`.

4.1 DataGene utility class

In the design of the optimizer, a series of general static methods are used, which are collectively placed in the `DataGene` class to enhance system flexibility and maintainability. The methods in this class cover multiple functions, mainly including the following six aspects:

1. Conversion between Timestamps and Strings
   The `time2str()` and `str2time()` methods implement mutual conversion between the `datetime` format and strings. Considering that the OCPP protocol requires WebSocket message timestamps in the format `%Y-%m-%dT%H:%M:%SZ`,

while the optimizer needs timestamps in `datetime` format for calculations, these methods effectively convert string-based time representations into timestamps, taking account of the influence of time zones and summer/winter time issues. To achieve this, the `pytz` library is used to handle time zones and summer/winter time issues, in combination with the `strftime` and `strptime` functions from the `datetime` library for time format conversion.

2. Format Conversion of Dictionary Keys
   The `snake_to_camel_case()` and `convert_dict_keys()` methods address inconsistencies in dictionary key naming, particularly the mixed usage of snake case and camel case commonly found in the OCPP library. These methods standardize dictionary key names to the format required by the optimizer, improving code readability and consistency.

3. Generate Electricity Price Array
   The `gene_eprices()` method is used to generate an electricity price array in the format required by the optimizer, ensuring accurate cost calculations during the optimization process. This method generates a list of 96 15-minute intervals, based on daytime and nighttime electricity prices and the corresponding switching times. In cases where nighttime prices or switching times are not provided, the system defaults to a unified price to accommodate different pricing strategies.

4. Split Time Period
   The `split_time()` method is primarily used to split electricity usage records, electricity prices, and grid power over a specified time period into segments, forming an array for the optimizer to perform optimization at the time period level. This functionality is particularly important when considering the variation in power demand across different time periods.

5. Generate User's Power Usage History
   The `gene_his_usage()` method uses `np.random.normal()` to generate random historical power usage records for a household, simulating fluctuations in electricity demand at different times of the day. The method is specifically designed with peak usage periods for day and night, and it also supports generating fixed historical usage records by setting a fixed seed, facilitating comparisons and experiments with different optimization modes.

6. Drawing Charging Graphs
   The methods `plot_usage()`,`plot_charging_curve()`,

`plot_usage_comparison()` and `plan2figure()` are used for visualizing different charging data. `plot_usage()` displays the historical electricity usage line graph; `plot_charging_curve()` shows the power variation curve during the electric vehicle charging process; `plot_usage_comparison()` compares the household's power usage before and after optimization with the maximum grid power, visually showing the optimization effect; `plan2figure()` provides real-time updates of the charging process, displaying the current and estimated charging amounts.

4.2 Optimizer class

The optimizer is one of the core components in CSMS . It's main function is to calculate and generate an optimized charging schedule based on the charging needs received through OCPP and other related data. One of the core targets of the optimization process is to achieve peak shaving, in order to improve the overall efficiency of the charging network, reduce operational costs, and ensure the stable operation of the power grid.

4.2.1 Optimization considerations

In the electric vehicle charging optimization problem, there are numerous challenges that need to be considered comprehensively, such as grid load, price fluctuations, battery health, charging energy sources, and user experience. This section proposes corresponding solutions to the key issues of charging optimization as following.

1. **How to predict future charging demand and balance grid load through dynamic scheduling.**
   Solution: Based on historical charging data, user behavior patterns, weather changes, and other factors, machine learning methods can be used to forecast future charging loads. In addition, dynamic scheduling algorithms (such as dynamic programming) can be employed to dynamically optimize charging strategies by integrating real-time data, avoiding charging peaks, and improving load balancing in the grid.
2. **How to respond to dynamic price fluctuations and optimize charging costs.**
   Solution: Since electricity prices are influenced by factors such as time, demand , and weather, machine learning predictive models can be used to predict future price trends. By training forecasting models based on historical price data, weather

conditions, grid load, and other information, charging scheduling strategies can be adjusted at different periods to optimize the user's charging cost.

3. **How to meet the charging preferences of different users.**
   Solution: Provide multiple charging modes to meet different user needs, including:
   - **Priority Charging Mode**: Suitable for users who urgently need power, completing the charging in the shortest time.
   - **Priority Saving Mode**: Uses the price prediction model to automatically schedule charging during low-price periods to reduce charging costs.
   - **Balanced Mode**: Considers charging time, electricity price, and grid load to optimize charging within a reasonable time frame.
4. **How to reduce battery aging and improve battery lifespan.**
   Solution: Battery aging is mainly influenced by the depth of discharge, charging frequency, and charging power. Therefore, intelligent charging strategies can be adopted, such as avoiding deep discharge, high-rate charging, and using slow charging when possible. In addition, optimizing the charging curve to ensure the battery charges within the optimal voltage range can reduce internal losses, thus extending the battery's lifespan.
5. **How to further reduce the load on the grid and achieve load balancing.**
   Solution: Using Vehicle-to-Grid (V2G) technology, electric vehicles are allowed to feed back some power to the grid during peak load periods, thereby reducing grid load. At the same time, V2G can charge during periods of low electricity prices or low loads, and supply power during peak hours, which can not only optimize grid load, but also bring additional benefits to car owners.
6. **How to improve the user's charging experience and enhance charging convenience.**
   Solution: Based on big data analysis, provide a personalized charging experience. By analyzing the vehicle owner's travel habits, historical charging records, electricity price fluctuations, and other information, the system can recommend the optimal charging time and location for the user. For example, it can intelligently suggest currently available charging stations, reducing wait times. Additionally, by integrating with the navigation system, the recommended charging stations can be adjusted in real-time to further enhance charging convenience.

Considering that the optimizer is not the focus of this project, we have used simple methods to implement basic optimization.

- For predicting charging demand, we use the `gene_his_usage()` method in section 4.1 to generate historical electricity usage records.
- For dynamic electricity price changes, we use the `gene_eprices()` method in section 4.1 to generate electricity prices, distinguishing only between day and night prices.
- For different charging demands, we added a new key `mode` in the `customData` of the OCPP message to record and adjust the weight of the parameters in the cost function inside the optimizer. A value of 0 represents dynamic adjustment, 1 represents the shortest charging time, and 2 represents the minimum charging cost.
- To enhance battery life, we have implemented a strategy of continuing charging with minimal current after the battery is fully charged, until the vehicle departs.
- For using V2G technology to further reduce grid load, since the vehicle used in this project does not support V2G functionality, it has not been considered in the optimizer design.
- Personalizing the charging experience is one of the features we hope to improve in the future. We plan to develop and refine a mobile app to make charging more convenient.

4.2.2 Optimization method

In this optimizer, we ultimately chose the `scipy.optimize.minimize` function for optimization. Compared to deep learning, it has the following advantages and disadvantages:

Advantages:

- The algorithm is simpler and suitable for rapid deployment.
- The computation efficiency is high, making it suitable for small-scale optimization problems, and it typically converges faster than deep learning methods.
- It does not require a large amount of data and can directly solve mathematical optimization problems without a training process.

- It has strong interpretability, with a transparent optimization process that allows tracking of objective function values, gradients, etc.
- It is suitable for short-term scheduling or online optimization.

Disadvantage:

- When charging optimization involves multiple uncertain factors (such as price predictions and user behavior), minimize may get stuck in local optima.
- Many minimize algorithms rely on gradient information, which may lead to optimization failure or slow convergence if the objective function is non-differentiable or non-smooth.
- Charging optimization may involve long-term decisions (such as predictive optimization based on user historical data), and minimize lacks learning ability, whereas deep learning can learn time-series patterns.

Finally, we transformed the charging optimization problem into a constrained nonlinear optimization problem, with the goal of optimizing electric vehicle charging schedules under different electricity prices and power constraints to minimize both time and cost.

Optimization objective:

$$\min_{P_1,P_2,\ldots,P_N} \quad w_1 T + w_2 C \tag{4.1}$$

Where:

- P_i: Charging power in the i-th time period (kW)
- N: Total number of time periods
- w_1, w_2: Weights for time and cost
- T: Charging completion time
- C: Charging cost

Charging Time:

$$\bar{i} = \begin{cases} \min\left\{ i \,\middle|\, \sum_{j=1}^{i} P_j \Delta t_j \geq E_{\text{target}} \right\}, & \text{if } \exists i \text{ such that } \sum_{j=1}^{i} P_j \Delta t_j \geq E_{\text{target}} \\ N, & \text{otherwise} \end{cases} \tag{4.2}$$

$$T = k \times \bar{i} \times \bar{\Delta} t \tag{4.3}$$

Where:

- E_{target}: Target charging energy (kWh)
- P_j: Charging power during the j-th time period (kW)
- Δt_j: Duration of the j-th time period (h)
- $\bar{\Delta} t$: Average duration of each time period (h)
- $k = 4$: Time conversion factor

Charging cost:

$$C = \frac{\sum_{i=1}^{N} P_i \cdot p_i \cdot \Delta t_i}{\bar{p}} \tag{4.4}$$

Where:

- P_i: Charging power during the i-th time period (kW)
- p_i: Electricity price during the i-th time period (EUR/kWh)
- Δt_i: Duration of the i-th time period (h)
- $\bar{p}$: Average electricity price over the entire horizon (EUR/kWh)

Constraints:

$$\sum_{i=1}^{N} P_i \Delta t_i \geq E_{\text{target}} \tag{4.5}$$

$$P_{\min,i} \leq P_i \leq P_{\max,i}, \quad \forall i \in \{1, 2, \ldots, N\} \tag{4.6}$$

Where:

- $P_{\min,i}$, $P_{\max,i}$: Minimum/Maximum charging power during the i-th time period (kW)

Initial Value:

$$P_0 = \frac{1}{N} \sum_{i=1}^{N} P_{\max,i} \tag{4.7}$$

To balance charging time and charging cost, we set both factors as parameters in the cost function. Since these two quantities have different units, a scaling constant k in Equation 4.2 is introduced to normalize their magnitudes, and weights are applied to adapt the cost function to three different charging strategies.

The initial value is set to the average of the maximum charging power over all time periods. This choice is based on the typical behavior under real-world electricity price schedules, where charging strategies tend to maximize power during low-price periods and reduce power during high-price periods. By using this initial value, the optimizer can converge more quickly, improving computational efficiency and reducing the risk of being trapped in extreme local minima.

4.2.3 optimization results

After using the optimizer, we got a satisfactory result. Below I will select one of the optimization results to show. The parameters used in this optimization are shown in Table 4.1.

Parameter	**Value**
Energy Amount (Wh)	12000
EV Max Current (A)	13
EV Min Current (A)	6
EV Max Voltage (V)	230
Start Time	16:00
Departure Time	4:00(+1 day)
Daytime(22-6) Electricity Price (Euro/KWh)	0.33
Night(6-22) Electricity Price (Euro/KWh)	0.3
Maximum Grid Power (Wh)	4000
Optimize Interval (min)	15

Table 4.1: Optimization Parameters

We optimized the three modes under the conditions in the table and obtained the following results. Figure 4.1 shows the change of the total charging amount over time in the three modes. It can be clearly seen that under the "shortest charging time" strategy, the curve reaches the target charging amount at the fastest speed, and under the "minimum charging cost" strategy, the car only starts charging 6 hours after the start of charging, that is, when the night electricity price starts, so it also achieves the minimum cost of 3.6 euros.

Figure 4.2, Figure 4.3, and Figure 4.4 respectively show the comparative changes in grid power after using the optimizer in three different modes. In general, the total power after all the car charging does not exceed the maximum grid power limit, and the optimizer perfectly completes its role.

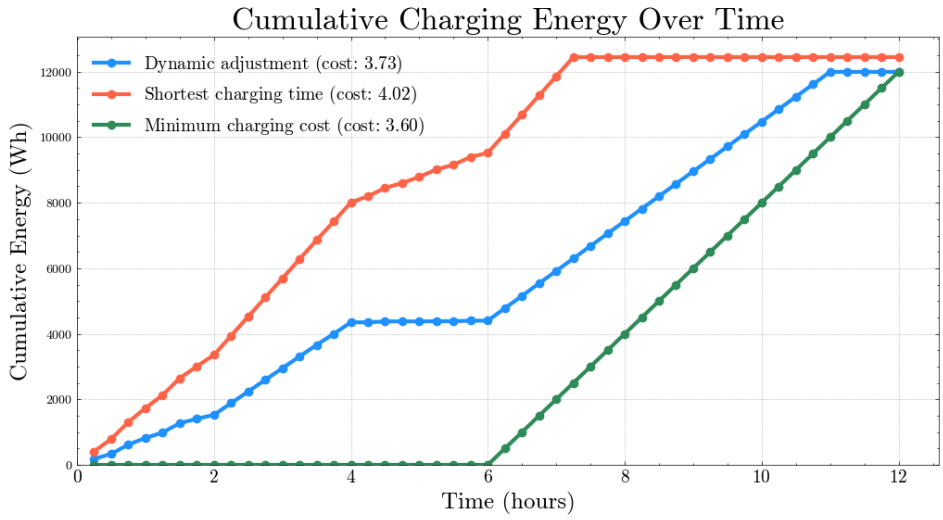


Figure 4.1: Charging comparison under different strategies

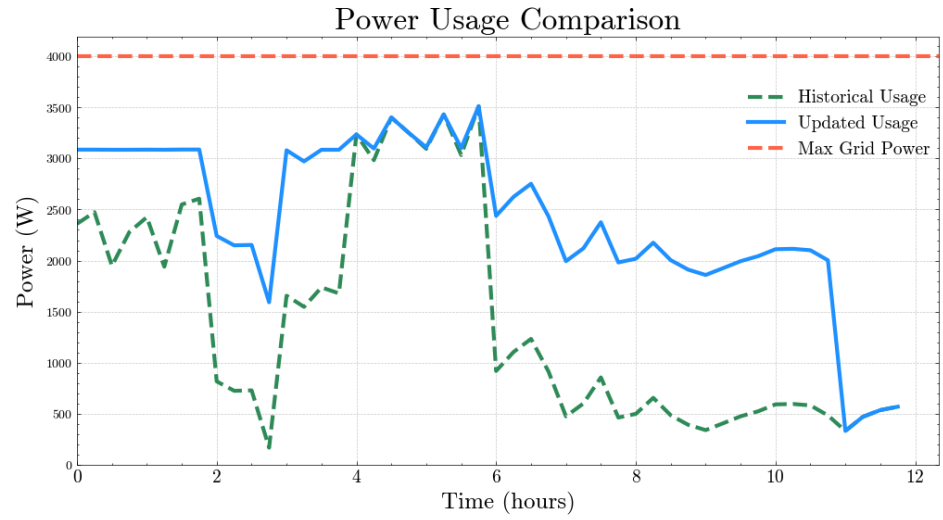


Figure 4.2: Power usage comparison with Dynamic adjustment

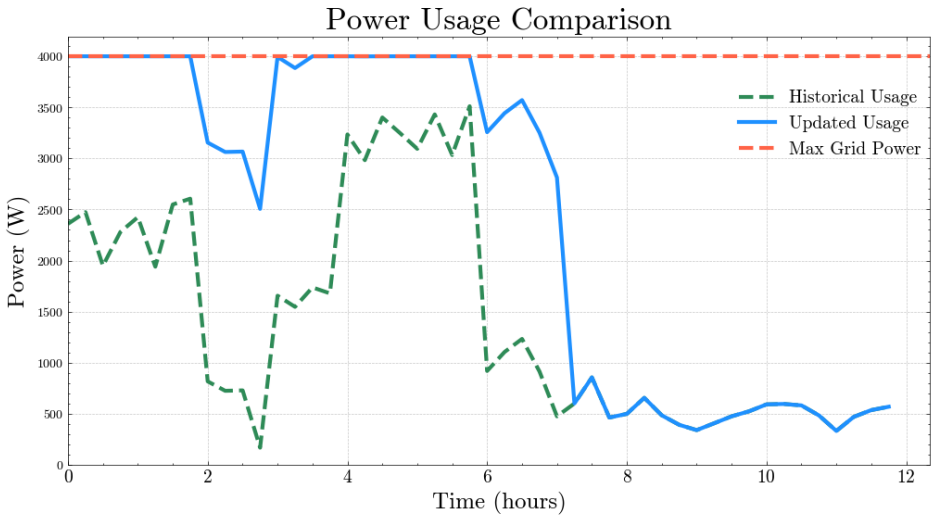


Figure 4.3: Power usage comparison with shortest charging time

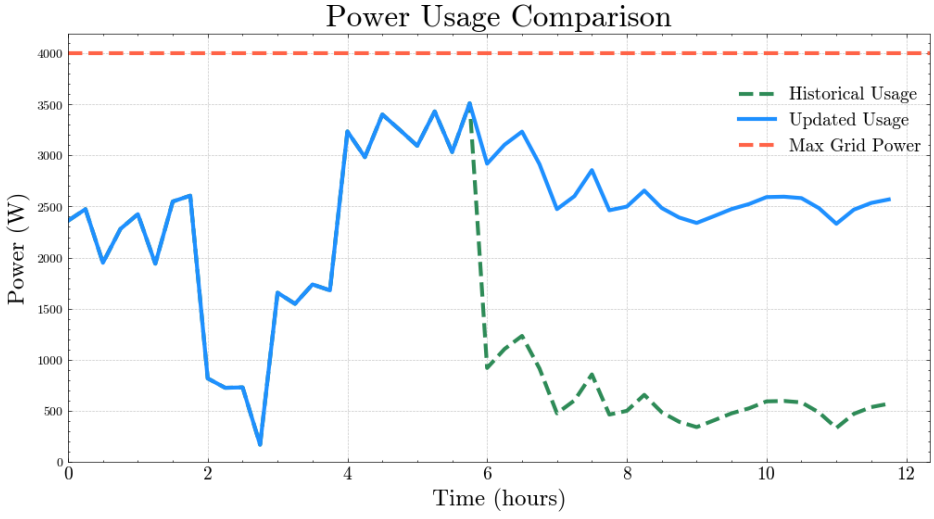


Figure 4.4: Power usage comparison with minimum charging cost

4.2.4 optimization problems

During the practical implementation of the optimizer, we encountered several situations that could lead to optimization failure:

- Currently, the optimization process assumes equal time intervals for each time slot. When the time intervals are not equal, especially when some time intervals are relatively short, the optimizer will fall into a local optimum, resulting in optimization failure.
- When the initial value P_0 is set as the average of $P_{\min}$ and $P_{\max}$ for each time slot, it may fail to satisfy the total energy constraint. This places the initial point outside the feasible region, causing the optimization to fail at the start.
- When dynamically adjusting the weights in the cost function to reflect different charging strategies, the gradient direction obtained in a previous optimization step may no longer be valid for the current iteration. This inconsistency can hinder convergence and ultimately lead to optimization failure.

4.3 Main function Server

In the main function of the CSMS, the most important task is to instantiate the threads and coroutines of each module, and bind different signals to their corresponding handler functions.

The main instantiated thread is the `PortWebServerOptimizer` class for web data visualization, and the coroutine is the `PortOCPPWebsocketServer` class for OCPP communication. Since the optimization algorithm itself is relatively simple and computationally efficient, it is not assigned a separate thread or process to run.

Therefore, in the main function, we mainly handle the following types of messages:

1. Website Messages: Primarily receive the data required by the optimizer submitted from the website. Once received, assign the data to variables in the main function and pass them to the optimizer.
2. Console Messages: Receive various messages from the console and forward them to the website for display. Specifically, identify and extract the IP address of successful `WebSocket` connections for use by web widgets.

3. OCPP Messages: Mainly handle the `NotifyEVChargingNeeds` message. Upon receiving it, pass the parameters to the optimizer for optimization. Once optimization is successful, generate a `SetChargingProfile` message and store it in the pending send queue. After receiving a successful response, remove the message from the queue. If a failure or timeout occurs, retry sending the message.

5 Charge Station

As the final execution part of the entire system, the CS implements complex functions such as OCPP communication and communication with EVSE and Shelly devices, as well as signal control. This chapter mainly introduces the charging execution logic, module design, and functions, including the charging unit module, charging data manager module, thread management module for hardware polling, Modbus communication module, and GPIO manager module.

5.1 Charge unit

This section mainly introduces the design and function of a single charging unit. A single charging unit can be understood as a charging socket on a charging point. It is the basic execution unit in the charging point and directly corresponds to the specific execution process of a charging task or plan. It should be noted that the charging unit is not responsible for the global control and management of the entire charging point, but only focuses on the single charging process associated with itself. It is mainly responsible for establishing a connection with the charging vehicle, performing charging operations by controlling the EVSE module, monitoring the state changes of the charging process, and synchronizing relevant data to the data manager and GPIO manager. In actual application, a charging point may contain multiple charging units, each of which is independent of the others. In theory, it can provide charging services for different vehicles simultaneously.

Definition of the words:

- **Charging unit**: A fundamental execution unit that consists of an EVSE and Shelly hardware, responsible for managing a charging socket. Each charging unit operates independently, executing charging schedules to control the start, stop, and monitoring of the charging process.

- **Charging schedule**: A structured JSON data set that defines at least one charging action along with charging-related parameters (such as charging unit, EVSE ID, charging start time, etc.). Its format corresponds to the OCPP message `SetChargingProfileRequest`, which specifies the execution strategy of the charging unit.
- **Charging action**: A discrete charging command, consisting of two key parameters: the charging action period and the charging limit. Each charging action corresponds to a single element in the OCPP message `SetChargingProfileRequest.chargingProfile.chargingSchedule.chargingSchedulePeriod`.
- **Charging action period**: The duration of a charging action, representing the time span from the start to the end of a specific charging operation.

5.1.1 Charge mode

In order to meet various application scenarios and testing needs, three charging modes have been designed.

- **Direct charging mode**:

  In this mode, the system bypasses any predefined charging schedules and directly delivers power according to the maximum current allowed. This mode is simple and direct, without time alignment, truncation, and periodic correction. It is suitable for scenarios with fast charging or no special time requirements. The charging process continues until the user manually ends it, and the system will not automatically stop charging.

- **Manual charging schedule mode**:

  In this mode, the system charges according to a user-defined charging schedule. Time alignment and correction are not performed in this mode, which is primarily intended for debugging purposes or scenarios where the optimizer cannot be connected. Charging is automatically terminated upon completion of the scheduled plan.

- **Dynamic adjustment mode**:

  Initially, the system will generate an OCPP request message based on the input data of the external management end and send it to the optimization server, and then charge according to the charging schedule returned by the optimizer. The schedule

includes information such as charging period and power limit. In the first cycle of charging, the system will time align the schedule to ensure accurate execution of the plan. This mode is suitable for scenarios where users have specific time or power requirements, such as peak and valley electricity price strategies or load management (Figure 5.1).

5.1.2 Smart charging logic

The preparation process before charging and the cleanup process after charging are the same across all three charging modes. The dynamic adjustment mode enables smart charging by generating a charging schedule through the optimizer and modifying it during the charging process. The core logic of the smart charging process includes the following steps: pre-charging status check (item A.), time synchronization in the charging schedule, charging time configuration during the charging process, periodic correction of the charging schedule (item B.), determination of charging completion, and the resetting and post-processing of the charging unit after charging is completed (item C.).

A. **Before charging**:

   Before initiating the charging process, the charging unit will first perform a series of pre-checks to ensure that the charging environment and device status meet the basic conditions for safe charging. The primary inspection objects include the EVSE module and the shelly module. The system will first confirm whether these two key hardware modules are working properly. Only when both are fault-free will it continue with the subsequent charging logic judgment. Once an abnormality or fault is detected in the EVSE or Shelly module, the charging setting process will be terminated immediately to avoid safety risks or equipment damage.

   Next, the system will detect whether the charging point has established a physical connection with the vehicle. If it is detected that there is no vehicle connected, the system will directly terminate the charging setting to avoid invalid or erroneous charging tasks.

   In order to further ensure charging safety and system reliability, an error protection mechanism is also introduced in the design. Considering the hardware errors that may have occurred during the last charging process (such as Shelly overload, EVSE connection interruption, etc.), directly entering a new round of charging may cause abnormal system status or unpredictable safety hazards. For this reason, an error

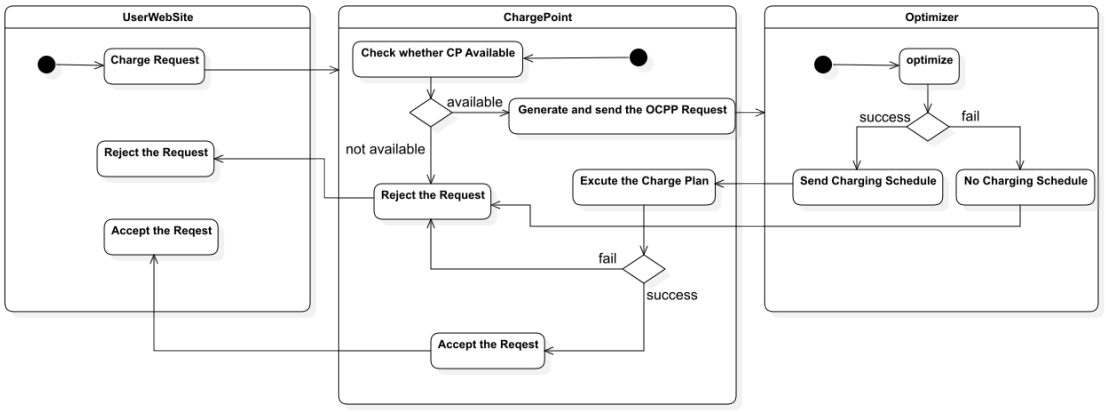


Figure 5.1: Charging Plan Request Process

flag is maintained inside the charging unit. When a hardware or system-level abnormality is detected, the flag will be set to `True`, and the system will reject any new charging tasks. The error must be confirmed and processed by the external management end (such as the web management platform). After troubleshooting and resolving the issue, the error flag must be manually cleared before the charge is resumed to ensure that the problem is discovered and handled in time.

After passing all the above safety checks and charging mode selections, the system will also perform a final round of status verification. This includes confirming that the charging unit is not already in a charging state, ensuring the EVSE has no vehicle error alarms, and verifying that the Shelly module is free from power abnormalities, to ensure that all operating environments are safe and stable. After completing these checks, the system will clear the charging data left over from the last charge to prevent historical data from interfering with this charge. Simultaneously, the relevant data and parameters for the new charging request are initialized and set, preparing the charging unit to begin the actual charging process. In the end, the charging point will lock the charging plug.

B. **In charging**:

After confirming that the charging preparation is complete, the charging unit needs to process and verify the charging schedule. Due to factors such as network latency and time synchronization errors, the execution time of the charging schedule may deviate from the current actual time, potentially resulting in the schedule being

too early or too late. To ensure the accurate execution of the charging process, the charging unit will first trim and synchronize the charging schedule. The specific logic will be explained later (subsection 5.1.3).

After all pre-processing steps are completed, the charging unit will execute each charging action cycle in sequence. When the planned execution time is later than the current time, the system will wait until the planned execution time arrives. In each charging cycle of the entire charging process, the charging unit will manage and record three important data items:

- **List of charging actions waiting to be executed**: includes all charging actions to be executed, and each charging action includes the relative time of charging start and charging limit.
- **Current charging action**: the charging action currently being executed.
- **Completed charging action list**: contains charging actions that have been executed, which can be used to draw charging curves and other analyses.

Whenever a charging action is completed, the system will move it to the list of completed charging actions and continue to extract the next charging action from the waiting queue and load it into the current execution item, repeating the above process until the entire charging plan is completed. To ensure the accuracy and stability of the charging process, the system will recalculate the duration of this charging action before executing each charging action. Due to the influence of program execution speed and external factors, the actual time interval may fluctuate. Therefore, rather than simply relying on the relative time period of the charging schedule, this project design uses Equation 5.1 to calculate the duration of the charging action to ensure that the start and end time of the charging action strictly conforms to the plan:

$$T_{duration} = t_{next} - t_{current} \tag{5.1}$$

Where:

- $T_{duration}$: The duration of the current charging action
- t_{next}: The absolute start time of the next charging action
- $t_{current}$: The current system timestamp

After calculating the duration of the current charging action, the system will simultaneously check whether periodic correction of the charging schedule is required (as detailed in subsection 5.1.4). After completing the correction, the charging unit will send an instruction specifying the charging current to the EVSE module to start the charging process.

It should be noted that the correction operation of the charging schedule only updates the content of the schedule and does not trigger the initialization process of the charging flow again (Figure 5.2).

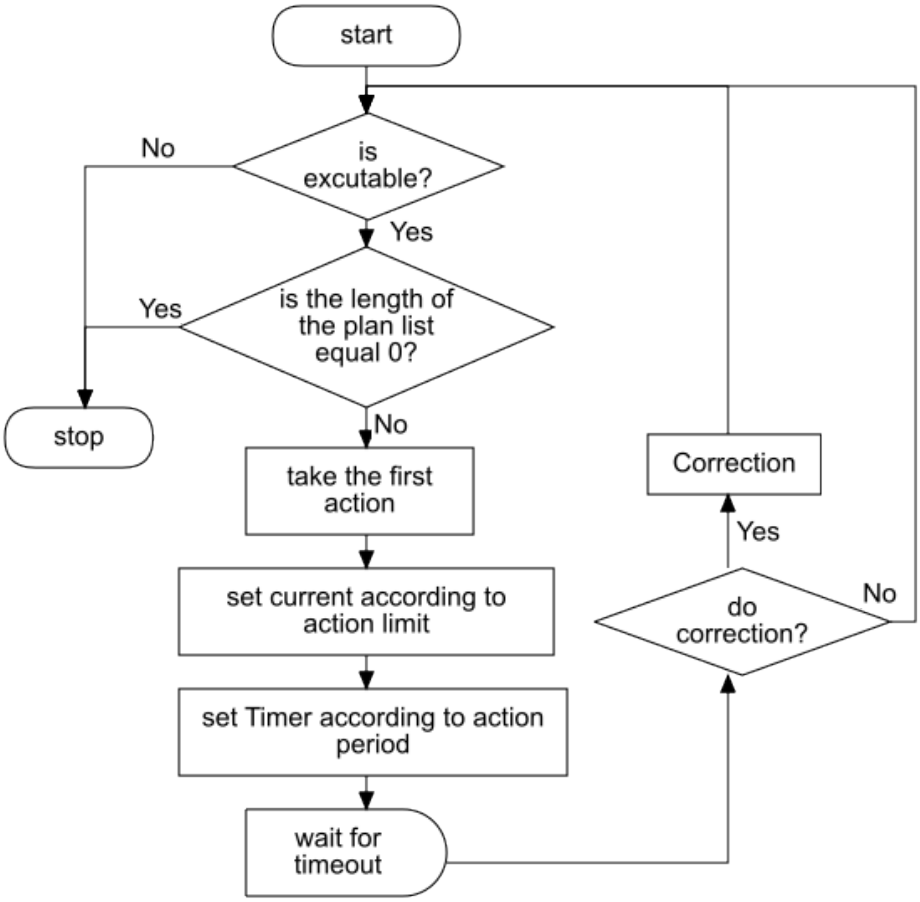


Figure 5.2: Charging Process

C. **After charging**:

During the charging process, the system will continuously detect the current charged energy amount to ensure that the charging process proceeds according to the predetermined plan. When the charged amount reaches the planned charging amount, the system will automatically terminate the charging. If the charging process is completed as planned, the system will perform a series of operations to end the charging.

First, the charging unit will send a stop charging instruction to the EVSE module to immediately terminate the charging action. Next, the system will perform the unlocking operation of the charging plug latch to ensure that the charging plug can be safely pulled out so that the user can take the charging plug away. Finally, the system will reset the flags of all charging units and the list of charging actions waiting to be executed. This ensures that the charging unit is restored to its initial state, preventing any historical data or charging status from interfering with subsequent operations.

In addition to the normal termination of charging, the system also provides an emergency termination mechanism. For example, when a hardware failure occurs or the user manually stops charging, the system will issue an emergency termination command through the external management terminal to quickly shut down the charging and reset the charging unit. This mechanism can protect the safety of equipment and users in emergency situations and ensure that the charging process will not cause further safety issues due to failures. All these mechanisms are designed to ensure the safety, reliability, and flexibility of the charging process.

Specifically, when a hardware failure occurs, the system will immediately set the charging unit's error flag to `True`, preventing further charging operations. The charging unit will be locked, and no charging tasks will be allowed until the fault is resolved. After the error is cleared and the flag is manually reset via the external management platform, the system will permit charging operations to be set again. This mechanism ensures that the system halts any charging processes that could lead to further damage in the event of a hardware failure.

However, if the user manually stops the charging operation, the system will not set the error flag to `True`, and the charging task can be reset. Manual interruption typically occurs for normal reasons, such as switching to charge another vehicle or adjusting charging parameters. In this case, the charging unit remains in a normal state and is not locked, allowing the user to reset the charging plan as needed.

5.1.3 Time synchronize and split

In actual operation, due to system response delays, network transmission delays, or other external factors, the actual execution time of the charging unit may lag behind the ideal execution time specified in the charging schedule. Additionally, considering that multiple charging stations may be operating simultaneously, directly executing the delayed charging plan without adjustments could easily lead to an instantaneous power surge that exceeds the system's design limits, resulting in safety risks or potential equipment damage.

Taking a typical scenario as an example, suppose three charging stations are scheduled to charge at 1 kW, 1 kW, and 4 kW respectively during the first time period. After three minutes, they enter the second scheduled time period, each charging at 2 kW. The system is designed with a total power limit of 6 kW. If, due to a delay, the third charging unit starts later than scheduled but still follows the original charging time and power plan, then after three minutes, the actual power usage of the three charging units becomes 2 kW, 2 kW, and 4 kW, totaling 8 kW. This clearly exceeds the power limit and is highly likely to cause equipment overload or trigger a system shutdown.

Therefore, to avoid the problems mentioned above, the system must "trim" the charging schedule before execution, removing the expired or no longer applicable schedule segments. This ensures that the actual charging power aligns with the current system capacity and safety design requirements. The trimming process is illustrated in the diagram (Figure 5.3). During this process, the system will automatically calculate the effective charging action that should be executed immediately at the current time point and use it as the new execution starting point to prevent power surges caused by delays.

Besides the trimming operation, the time alignment of the charging plan is also crucial. In the actual charging process, there is usually a deviation between the current system time and the planned charging action execution time, especially when multiple points are running in parallel, there are delays in network transmission, or the system processing speed fluctuates. This deviation will be more obvious. If effective time alignment is not performed, the actual charging process may deviate seriously from the plan, which will affect the scheduling efficiency and power control accuracy of the overall system.

In addition, before officially starting charging, the charging unit must complete a series of safety checks and preparations, such as EVSE module self-inspection and closing the charging plug latch with the DC motor. These operations will take time. If the charging plan is executed directly, it will not be possible to ensure that the equipment is in a fully prepared and safe state.

According to the system design, if the EVSE self-test function is enabled, the EVSE self-test process takes about 30 seconds, and the start-up time of the latch motor is expected to be 2 seconds. Therefore, before the system officially enters the first charging action cycle, the overall planned time needs to be postponed by 32 seconds to fully reserve the above preparation time to ensure that the various functional modules of the charging unit are in normal status and achieve controllable and safe operating conditions. The adjusted time alignment calculation formula is as follows

$$t_{syn} = t_{start} + t_{rel} - (t_{evse} + t_{motor}) \tag{5.2}$$

Where:

- t_{syn} : actual execution time after synchronization
- t_{start} : absolute planned execution start time
- t_{rel} : relative time of the first charging action executed in the schedule
- t_{evse} : EVSE self-test time
- t_{motor} : latch motor running time

After obtaining the actual execution time t_{syn} after synchronization, the system will make judgments and processes based on the current system time $t_{current}$:

- If $t_{current} \leq t_{syn}$, it means that the current time has not yet reached the actual execution time of the charging action, and the system will enter a waiting state until the synchronization time point is reached before starting to charge.
- If $t_{current} > t_{syn}$, it means that the current time has exceeded the scheduled actual execution time. In order to ensure the continuity and accuracy of the charging plan, the system will subtract this time difference $|t_{current} - t_{syn}|$ from the duration of the first charging action to avoid misalignment of the charging action, and then immediately execute the charging plan.

Through the time alignment mechanism mentioned above, the system can effectively address the time deviation issues caused by preprocessing tasks (such as EVSE self-test, and charge plug locking) or other execution delays. Once the time alignment is completed, the charging unit will strictly follow the adjusted time sequence and plan parameters during the execution of the entire charging plan. This not only ensures the accuracy and stability of the charging process but also guarantees that the dynamic changes in charging

power adhere to the system's scheduling and safety strategies, thereby preventing risks such as power overload due to time drift or plan deviation.

Additionally, the trimming and time alignment mechanism of the charging plan is designed to be executed each time the charging plan is set, as follows:

- When the charging plan is loaded for the first time, the system will automatically perform trimming and alignment, remove any expired historical charging actions, and adjust the remaining plan to align with the current system time.
- Each time the charging plan is corrected or updated, the mechanism will be triggered again to re-trim and realign the plan, ensuring that the corrected plan seamlessly integrates into the current charging process.

This design ensures the accuracy of the charging process and also reserves good expansion space for possible subsequent dynamic adjustment, load management, and multi-point collaborative control functions.

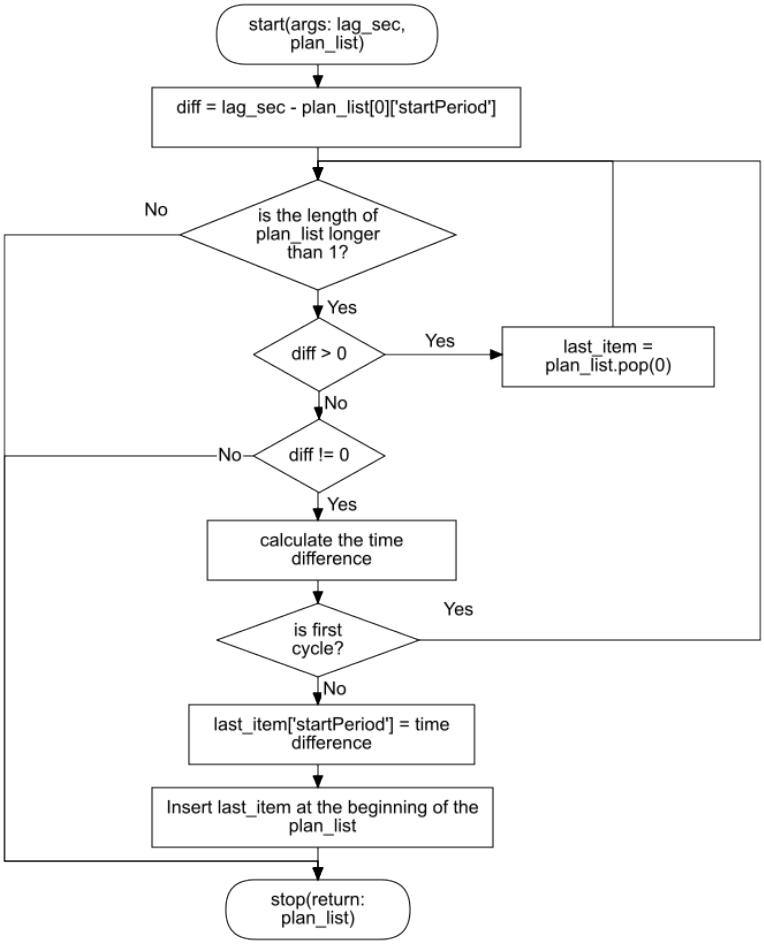


Figure 5.3: Time synchronize and split

5.1.4 Charge plan correction

During the charging process, the load power and charging current inevitably fluctuate, resulting in the actual charging capacity being different from the planned charging capacity. Without timely feedback correction, overcharging or undercharging may occur, thus affecting the precise control of the charging process, which in turn reduces the effectiveness of peak shaving. Therefore, it is crucial to periodically correct the charging

plan to ensure that the charging process is consistent with the plan and prevent the system from experiencing power deviation due to error accumulation.

In this system, the correction cycle is configurable, and the default setting is one correction per hour. Specifically, the correction mechanism is triggered by monitoring the system timestamp. When the current timestamp is floor divisible by 3600, meaning it reaches an exact hour, the system calculates the correction cycle index and compares it with the previous cycle index. If the index changes, which indicates that an hour has passed, the system will automatically trigger the correction of the charging plan. Notably, for the first charging cycle, the system will choose to skip correction when approaching the hour to prevent premature correction from interfering with the initial charging process.

The correction request process is as follows:

- **Trigger Condition**: Each time a single charging action is executed, the system checks whether the current timestamp meets the exact-hour condition (i.e., floor divisible by 3600). If met, the correction operation is triggered.
- **Request Transmission:** The correction request is transmitted via `GPIOManager`. The signal carries data containing the charging request and required parameters, which is sent from the logic module of the charging unit and further relayed to the optimizer through the main logic module. The system sends a `NotifyEVChargingNeedsRequest` message to notify the optimizer, requesting a corrected charging schedule.
- **Optimizer Response**: Upon receiving the request, the optimizer processes it through its internal algorithm and returns the latest charging schedule, based on which the system makes adjustments for the next charging phase.

Since a single charging point may contain multiple charging units, each of which needs to send a correction request to the optimizer, a large number of concurrent requests may be generated at the exact-hour timestamp. If these requests are sent simultaneously, the optimizer may experience response timeouts, or the OCPP sender may fail to transmit messages concurrently, resulting in some correction requests not being processed. Although this issue does not affect the ongoing charging operation and the request will be retried in the next correction cycle, it will cause the current correction attempt to fail.

To address this issue and prevent system bottlenecks caused by request conflicts, this project employs a message queue for request flow management. The message queue allows requests to be queued and sent in an orderly manner at predefined time intervals. This mechanism effectively prevents message congestion when multiple charging units

send requests simultaneously, ensuring that each unit's correction request is processed in a timely manner while also reducing potential system load pressure during high concurrency scenarios.

Specifically, the message queue is managed internally within the charging point. When multiple charging units trigger correction requests, the queue distributes these requests sequentially. A predefined interval is set between each request to prevent excessive request surges in a short period, thereby optimizing the overall system load distribution. This approach allows the optimizer to handle the distributed requests more efficiently, reducing response delays caused by excessive concurrent requests.

Through the implementation of the above-mentioned mechanism, the system can timely correct the charging schedule in each correction cycle, ensuring the stability and safety of the charging process. This effectively prevents charging issues caused by power imbalances or deviations in the charging schedule.

5.2 Data manager

Although this project does not explicitly require the use of a database, to support potential future historical data analysis, facilitate unified data management across modules, and ensure data consistency, this project uses a data manager. The data manager follows the singleton pattern, meaning that only one instance exists within the system, ensuring global accessibility and data uniqueness. All modules in the system can easily access and update the data, avoiding the problem of data asynchrony between multiple instances.

The main functions of the data manager include:

- **Charging unit data recording**: The data manager is responsible for recording the relevant data of each charging unit, including the data of the EVSE module, and the Shelly module. Every time a charging operation occurs, the system logs relevant information for future queries, analysis, and visualization.
- **Periodic generation of diagrams for GUI display**: In order to facilitate users and managers to view the charging status in real-time, the data manager will periodically generate diagrams related to the charging status and transmit them to the GUI for display. These diagrams intuitively depict charging data changes, allowing users to assess the charging process at a glance.

- **Data consistency assurance**: By adopting the singleton pattern, the data manager ensures the data consistency of all charging units. Whether it is the update of the charging schedule, the execution of the charging action, or the various real-time data generated during the charging process, the data manager can ensure the unified management of this information to avoid potential problems caused by inconsistencies between multiple data sources.
- **Historical data storage and analysis**: Although the current project does not directly require database support, the data manager provides a foundation for future historical data storage and analysis. If needed, it can support charging behavior analysis, strategy optimization, and report generation, ensuring scalability and adaptability.

Through this design, the data manager not only supports real-time charging operations but also provides a scalable framework for future system enhancements, data analysis, and expansion.

5.3 ModBus Communication

This project adopts the ModBus protocol as the primary communication method with underlying devices, such as EVSE modules and power metering equipment. Modbus adopts a Master-Slave architecture and supports multiple communication methods, including Modbus RTU, Modbus ASCII, and Modbus TCP[12]. Modbus uses serial communication, which transmits data bit-by-bit through one or multiple data cables. Typical baud rates are 9600/19200/115200 bps. Almost all industrial devices, such as PLCs, HMIs, and EVSEs, support Modbus. Additionally, using RS-485, can communicate with up to 247 devices. If in further cases, there is a demand for communication with multiple CSs, it's also extendable.

Within the Modbus communication protocol, different function codes can be used to read and write one or multiple registers simultaneously. It is worth noting that in this project, the EVSE supports only function codes 03 (Read Holding Registers) and 16 (Preset Multiple Registers). Therefore, communication with the EVSE is primarily based on these two functions.

Since its communication model uses a master-slave polling mechanism, it does not support active data push or real-time detection of connection disconnections. Therefore, to ensure

the validity of connections and the reliability of data during communication, the system design must explicitly define a connection management strategy.

For Modbus connection management, this project evaluated two mainstream design approaches and made a balanced choice based on actual requirements:

- Long connection in singleton pattern with thread polling for connection validity detection.

  This approach implements a Modbus connection as a singleton, ensuring that there is only one Modbus communication connection globally. The advantages include:

  - Connection reuse, avoiding repeated creation and destruction, thereby improving communication efficiency
  - Global accessibility, allowing any module within the system to directly call the communication interface

  However, since the Modbus protocol does not support active disconnection detection, a long connection requires a dedicated background thread to periodically poll or send heartbeat commands to check whether the device is offline. This increases the cost of thread maintenance, especially in scenarios with concurrent communication across multiple devices, potentially leading to high thread resource usage and memory accumulation risks.

- Short connection under a context manager.

  This approach encapsulates each read/write operation as a complete connection process using a context manager. During each communication operation, the system automatically establishes a connection, completes data transmission, and then immediately releases the connection resources. This avoids the issues of uncontrollable connection states and complex thread management associated with long connections.

  Although the short connection model introduces additional costs due to the frequent establishment and release of connections, the connection cost of Modbus itself is relatively low. Therefore, it is entirely acceptable.

Based on the above comparison and evaluation, the project ultimately decided to adopt the short connection mechanism under a context manager as the ModBus communication management strategy.

5.4 Thread Management (Polling)

Since EVSE and Shelly devices cannot actively push real-time data to the system, the system needs to adopt a polling mechanism to periodically retrieve their status information. However, considering that a charging point may contain multiple charging units, and each charging unit corresponds to a set of EVSE and Shelly devices, creating a separate thread for polling each EVSE and Shelly would not only increase the complexity of thread management but also lead to higher CPU usage and memory consumption, which would impact the overall performance and responsiveness of the system.

To solve this problem, the project designed an efficient polling mechanism that uses only two independent threads, one for EVSE data collection and one for Shelly data collection. The EVSE polling thread maintains a list of all EVSE devices and sequentially accesses the devices in the list according to the set polling cycle, reading the corresponding data and storing it in the data manager. The Shelly polling thread works similarly, maintaining a list of Shelly devices and periodically polling each device and storing the data. When modules like ChargePoint need to access data from EVSE or Shelly, they do not communicate directly with the devices but instead query the data manager for the latest polling data. This approach not only avoids the resource overhead caused by multiple threads concurrently accessing devices but also ensures unified data management, improving the system's stability, maintainability, and operational efficiency.

5.5 GPIO Manager

Due to the possibility that a charging station contains multiple charging units, efficiently managing these units is crucial for the stable operation of the system. This project uses the GPIO Manager to centrally store and manage the data of all charging units and their associated EVSE and Shelly modules. The system accesses each charging unit through the GPIO Manager and dynamically adjusts its operating state. This architecture effectively decouples the main logic of the system from the underlying hardware management and execution logic, so that the main program does not need to directly handle the specific communication and control logic of each charging unit, thus reducing the complexity and maintenance costs of managing different charging units.

In addition, the GPIO Manager is responsible not only for storing and scheduling device data but also for load balancing and peak-shifting of the charging unit's correction requests,

preventing message blocking that could occur from multiple charging units initiating correction requests simultaneously, ensuring the stability of the system and the reliability of data synchronization (subsection 5.1.4).

6 Front-end Data Display

Relying solely on physical buttons on the charging station makes it difficult to achieve precise selection and management of multiple charging units and various charging modes. Therefore, the system is designed with an external management GUI client, which provides basic information display and management operations of the charging station in the form of a web interface, making it easier for users to interact and control. Additionally, the system has reserved a WebSocket interface, allowing for future support of remote access and control of the charging system by other GUI clients.

Currently, the system's requirements are relatively simple, primarily implementing basic charging management functions, and have not yet involved more complex system management needs, such as changing OCPP protocol versions, dynamically registering or removing charging units at runtime or powering on/off the system and modules. However, to facilitate future expansion, the system architecture has reserved the potential for these functionalities, enabling future iterative development based on the existing framework.

6.1 Web Implementation

The web interface for this project is implemented through a web server built using Flask, with communication between the front-end and back-end primarily relying on HTTP. Currently, the system uses HTTP rather than HTTPS, meaning that the communication on the web does not include data protection or encryption features. Compared to HTTPS, this has certain security vulnerabilities. Since HTTP does not perform authentication, users cannot verify whether the server they are connecting to is legitimate. Additionally, HTTP transmits data in plaintext, making it susceptible to eavesdropping or tampering. To improve security, HTTPS encryption communication could be considered in the future to prevent data leaks and man-in-the-middle attacks.

In terms of front-end and back-end interaction, the system uses Socket.IO for bidirectional data message transmission, which enables smooth real-time communication. Operation commands such as starting or stopping charging are sent from the front-end web interface to the back-end. Meanwhile, the back-end pushes real-time status information to the front-end, such as the current charging status, system logs, and charging plan execution details.

6.2 Web Functionality

6.2.1 Charging Station Web Interface

The web GUI of the system is mainly divided into four sections, each responsible for different functions, as detailed below:

- **Home Page**:
  The home page is primarily used for selecting the charging mode and providing a basic information input interface for sending a request to the optimizer to obtain the charging schedule. To ensure the correctness of the user input, the system performs checks on the entered information, including verifying whether mandatory fields are filled, whether the data types are correct, and whether the input values are within a reasonable range. If any errors are found in the input, the interface will provide corresponding prompts and prevent the request from being submitted. The system will only allow the user to send a request to the back-end when all data is correct. Additionally, the home page provides control buttons for starting and stopping charging, allowing the user to manually control the execution of the charging process.

  The home page also features a language switch function, allowing users to select different display languages according to their needs. Additionally, the current server time is displayed on the interface, enabling users to judge whether the server is responding normally, and it also serves as an indicator of whether the WebSocket connection is disconnected.

- **Terminal Message Display**
  This section is primarily used to display the system's log messages, including changes in charging status, communication information, fault prompts, etc., allowing users to monitor the system's operation and diagnose faults. In addition to displaying

the logs, this section also provides a button that allows users to manually reset the system's error flags. This feature enables developers to intervene in the system's state and restore it to normal operation in case of errors.

- **Data Display**
  This section is used to display all data, including EVSE register data, Shelly measured data such as current, voltage, power, etc., and images of the charged amount, including a comparison of the predicted charge amount, calculated charge amount, and the actual charge amount measured by Shelly. If the charging schedule is generated by the optimizer, the system will also display the charging plan chart provided by the optimizer, allowing users to compare the actual charging situation with the planned data. In addition, in order to facilitate the recording and storage of data, we also provide a download button for downloading a CSV file containing the charging schedule and Shelly measurement data.
- **Manual Input of Charging Schedule**
  This section allows users to manually upload a custom charging schedule in CSV file format. To help users correctly fill out the file, the system provides a sample file as a reference. After the user uploads the file, the system will perform basic checks for missing values. If errors are detected, the system will display error messages on the interface, and the user can make corrections based on the prompts and re-upload the file.

6.2.2 CSMS Web Interface

The design of the optimizer's web interface focuses on simplicity, ensuring that data is displayed clearly at a glance. As shown in Figure 6.1, the main layout adopts a grid distribution, where a 2×2 grid divides the entire page into four distinct functional display sections:

- The top-left section is for **parameter submission**.
  This section is mainly used to fill in the parameters required by the optimizer, including the maximum grid power, optimization interval and electricity price. The electricity price only accepts csv file submission. Additionally, a validation check is implemented in the JavaScript file to ensure the submitted data is reasonable. When all submitted data is correct and successfully submitted to the CSMS main function, the webpage submission button will show successful submission, and the corresponding changes will also be displayed in the console message display part.

- The top-right section displays **charging demands**.
  This part is mainly used for displaying data after CSMS receives the charging needs report in the OCPP protocol sent by the charging point. A WebSocket connection prompt box is added in the upper right corner. When CSMS has no charging point connection, it displays "no connection". When a charging point is successfully connected, the IP address of the current charging point will be displayed.
- The bottom-left section shows **console messages**.
  This section is mainly used to display log messages, optimizer progress, data changes, and error prompts during the operation of the CSMS system. This section can greatly improve the speed of error troubleshooting.
- The bottom-right section presents **charging-related images**.
  This section is primarily used to display images after the optimizer has successfully completed optimization. It contains two main charts: the upper chart illustrates the charging capacity of the charging point over time after optimization, while the lower chart compares power consumption before and after charging. It also includes a reference line indicating the maximum grid power.

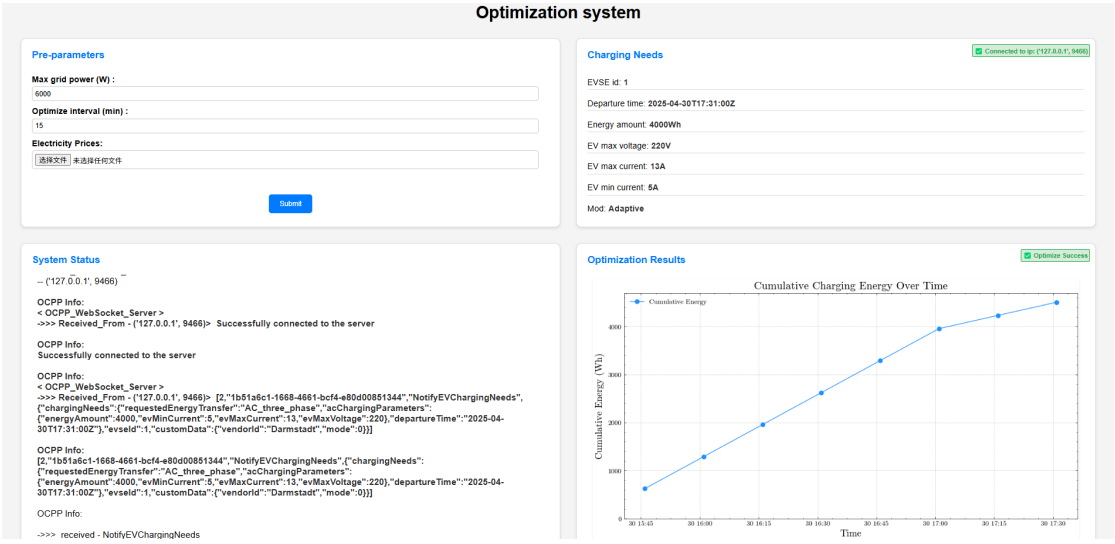


Figure 6.1: CSMS front-end page

7 Simulation and test

7.1 Simulator

Since actual charging control involves high-voltage electrical systems, there are significant safety risks. If the system has structural defects or code errors, performing live tests without sufficient testing may lead to component damage or even more serious safety accidents. Therefore, before performing actual power-on tests, mode testing must be conducted using a simulator to comprehensively verify the system's functionality. This ensures that the system's operating logic is correct. Only after the simulation tests pass should the actual power-on experiments be conducted, minimizing the risks as much as possible.

Simulator Structure The simulator in this project is mainly used to test the interaction and control between the EVSE and Shelly modules under a single charging unit. Therefore, the simulator is divided into two main parts (Figure 7.1):

1. EVSE Module and Motor Module
   Since the communication of EVSE and lock motor in the system is based on the Modbus protocol and relies on physical hardware implementation, during the simulation tests, software methods are used to simulate Modbus communication for system functionality verification. In the simulation environment, the system's hardware communication method is replaced with JSON file read/write. The simulator modifies the contents of the JSON files to simulate data changes of the EVSE device and motor.

   Specifically, when the system reads EVSE data, the original method of accessing the EVSE registers via Modbus is replaced by reading a JSON file. For example, when the charging unit writes a value of 0 to the 1004 register of the EVSE, it indicates that the EVSE begins supplying power to the vehicle. In this case, the value of the

key 1004 in the JSON file will be set to 0. Conversely, when charging is stopped, the value is changed to 1. The data read/write rules for other registers follow a similar approach. Additionally, the simulator is responsible for displaying certain register data that can only be modified by the EVSE, allowing real-time feedback on the EVSE state changes.

2. Shelly Module
   The Shelly module in this project communicates over WiFi using the HTTP protocol. As a result, the simulator must emulate an HTTP server to interact with the system. During the simulation of the Shelly module, the server periodically sends voltage and current data from the Shelly device, simulating changes in electrical parameters throughout the charging process. To better replicate real-world conditions, the simulated data does not remain fixed but fluctuates randomly around a set baseline value, reflecting the slight variations in voltage and current that occur during actual operation.

   The system can configure the simulator's IP address so that the program can correctly access the simulation server, enabling the simulation of communication with the Shelly device.

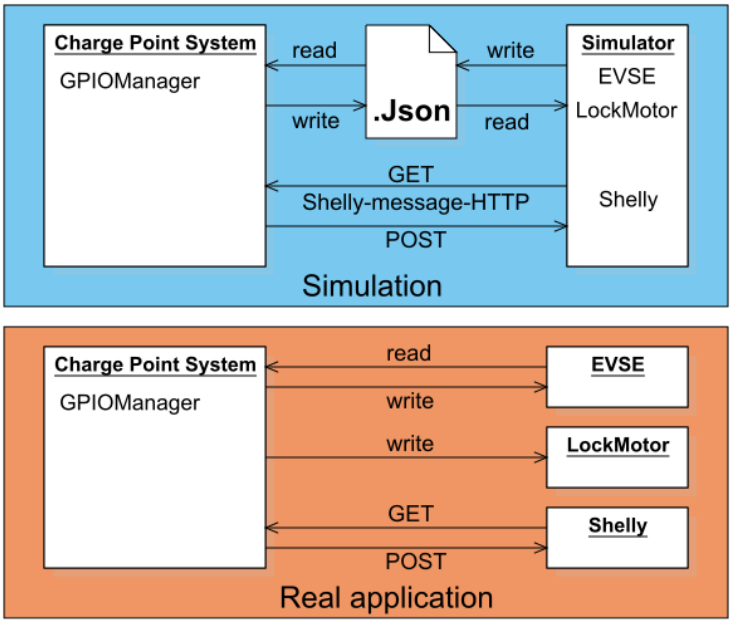


Figure 7.1: Simulation VS. reality

Simulator Implementation The simulator is developed using PyQt5 and provides a visual interface, allowing users to view and adjust simulation parameters. To facilitate distribution and usage, the simulator is packaged using PyInstaller to generate a standalone executable file, enabling it to run directly without the need for an additional Python environment. The packaging tool used for this project is PyInstaller-GUI (Github-Repo), which leverages PyInstaller for packaging (Figure 7.2).

This simulator allows for the debugging and optimization of the system's charging control logic in a safe testing environment, ensuring functionality correctness and reducing the safety risks associated with direct live testing.

Figure 7.2: GUI of Simulator

7.2 Charging Point Assembly and Testing

After completing the software simulation, it is crucial to perform charging tests (Figure 7.3). IMS provides an E-Vehicle that supports a maximum charging current of 30A. This vehicle can be controlled to start and stop charging via a remote controller after the charging cable is connected, which can also serve as an emergency stop method.

Figure 7.3: Conduct on-site EV charging station testing

On the CS side, there are two charging modes that need to be tested. The first mode is charging at a fixed rate using the `Start` and `Stop` buttons on the CS. The second mode involves controlling the charging process through the previously mentioned GUI. In this case, if an emergency stop is required, it can also be achieved by pressing the `Stop` button on the CS.

The problems occurred in testing:

1. At the beginning, there was no indicator light to show whether the CS was ready for charging, making it difficult to sense its status. Therefore, for better visibility, an LED on the `Start` button is connected to a transistor and controlled via a GPIO pin on the Raspberry Pi. If the LED is always lit, it indicates that the Raspberry Pi has finished starting up and is ready for the next step. At this point, the charging current can be set through the GUI. Once the current value is confirmed, the LED will start blinking, indicating that charging has begun.
2. The `Modbus` port can only be accessed by one thread at a time. If multiple threads attempt to access it simultaneously, resource contention issues arise, such as reading and writing to multiple registers at the same time. To solve this, we first use a thread lock to force multithreaded access into serialized access. Then, we apply the singleton pattern to ensure a globally unique access point. Furthermore, to manage this lock more effectively, a context manager is used.

3. Register 1006 consistently showed a bit status of 3 during charging, indicating a charging failure. According to the datasheet, register 1006 is read-only and cannot be modified. However, the documentation contains an error—register 1006 is actually both readable and writable. After setting register 1006 to status 1, the vehicle was able to charge normally.

8 Results & Discussion

8.1 Results

In this project, we have successfully achieved the following tasks:

1. **Charging point Control and Optimization**
   The control of the charging point is based on a charging schedule, which can be dynamically adjusted according to different strategies, such as dynamic charging, shortest-time charging, and lowest-cost charging. This enables the system to select the optimal charging time slots, reducing the burden on the power grid.

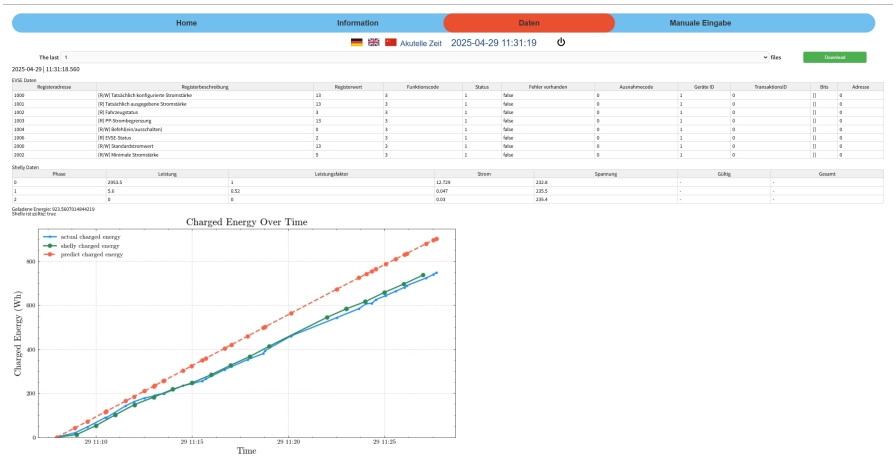


Figure 8.1: Execution of Charging Schedule

2. **Communication Mechanism Between Charging point and Optimizer**
   We have implemented a communication mechanism based on the OCPP protocol, allowing the charging point to interact with the optimizer for intelligent charge management and real-time data updates.

3. **Simple data visualization**
   In order to facilitate the viewing of the charging process, we have realized simple data visualization, including the charging capacity curve and charging power plan curve, power change curve during charging, household power consumption, and maximum power of the grid before and after optimization.

4. **GUI Design**
   The project provides a simple and user-friendly Graphical User Interface (GUI) to display charging status, log information, and control operations. This interface supports basic user interactions such as starting/stopping charging, selecting charging modes, submitting charging schedule requests, executing manually uploaded schedules, and remote management, making it easier to monitor and control the charging point.

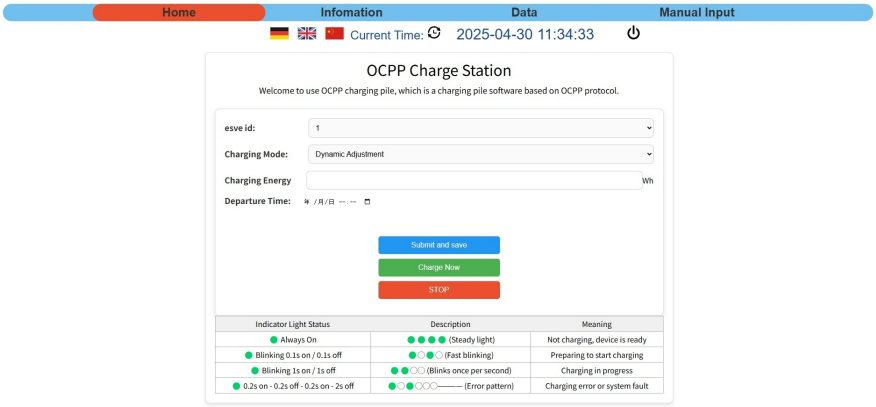


Figure 8.2: Graphical User Interface

5. **Software Simulation of the Charging Process**
   To reduce dependency on real hardware during development and testing, as well as to enhance safety during power-on testing, we implemented a software simulation of the charging process. This simulation system can emulate real charging scenarios,

including before charging, in charging, and after charging. It also generates corresponding status data, allowing for debugging and optimization in an experimental environment.

6. **Charging point Modification and Optimization**
   This project involves not only software system development but also hardware modifications and optimizations of the charging point to improve its compatibility and intelligence (Figure 8.3). The modifications include:
   - Using EVSE for charging control.
   - Adding a Shelly sensor to monitor charging current.
   - Control EVSE and Shelly through RaspberryPi with the script to achieve smart charging.

Figure 8.3: Charging Station Hardware

8.2 Discussion

This section summarizes some key issues and improvement suggestions for the current system in terms of functional design, technical implementation, and future optimization directions.

1. Concurrent Message Processing Issues
   Currently, the system has certain limitations in handling concurrent OCPP messages. As mentioned in section 3.3, the official OCPP Python library does not support concurrent message processing. Consequently, when the system needs to handle multiple OCPP requests simultaneously, it may lead to blocking or delays, affecting

the real-time control capability of the charging station. If the system requires high-concurrency OCPP message interactions in the future, the solutions provided in section 3.3 can be referenced to optimize the OCPP processing logic and enhance message processing concurrency.

2. Current Monitoring and Safety Control
   Currently, the system does not monitor or limit current parameters during the charging process, which may lead to the following issues:

   - Current Fluctuations: In certain situations, the current may experience significant variations, affecting charging stability.
   - Current Surges: A sudden increase in load may cause a sharp rise in current, which potentially affects the safety of the power supply system.

   To enhance safety and reliability, current monitoring and data filtering mechanisms can be added to the Shelly module in the future. For example, a threshold for current fluctuations can be set. If the fluctuation exceeds the predefined range, an alarm notification will be triggered, and the maximum current will be limited to the threshold value. In the event of an abnormal current surge, a protection mechanism will be activated to automatically stop charging, preventing equipment damage or potential safety incidents.

3. Multi-Charging Port Image Display Support
   Currently, the charging status image display function on the web front-end does not support dynamically adding or removing charging ports. The current UI is only applicable to a fixed number of charging units. If future requirements include dynamic management of multiple charging units, the front-end will need to be adapted accordingly.

4. Dynamic Registration of EVSE and Shelly
   Currently, the registration of EVSE and Shelly is implemented through an enumeration class. The drawback of this approach is that adding or removing charging units requires modifying the code and restarting the system. To enhance flexibility, a dynamic registration mechanism based on files or a database can be introduced in the future.

5. Limitation on Web Terminal Message Display
   Currently, the web front-end does not impose any limits on message display. If the system runs for a very long period and the webpage is not refreshed for a long time, it may lead to memory overflow or browser crashes. To prevent this issue, a

restriction should be applied to the number of displayed messages, while detailed logs can be accessed through log files.

6. Raspberry Pi WiFi Hotspot Support
   Currently, the system runs on a Raspberry Pi, but the existing Raspberry Pi model cannot set up WiFi hotspot. This means that when the WiFi connection on the Raspberry Pi is unstable or cannot be connected, the system management and remote access become inconvenient. If future Raspberry Pi hardware or the system supports the WiFi hotspot function, enabling AP mode on the Raspberry Pi could be considered, allowing users to connect directly to the Raspberry Pi for management.

7. Compatibility Issues with the OCPP Python Library
   Currently, the OCPP Python library used in the system has compatibility issues, particularly with the enumeration classes. Due to the slow updates of the official library, in order to better adapt to future versions of OCPP, it may be necessary to modify the OCPP library code or fully customize the OCPP library to meet specific business requirements.

8. Expansion of CSMS Functions
   Currently, the CSMS only implements basic charging management functions and does not include more comprehensive management features, such as:

   - **Database Support:** To record historical charging data for subsequent data analysis and optimization.
   - **Multi-Charger Management:** The current system is designed for the management of a single charging point. If future support for multiple charging points is required, the CSMS architecture will need to be expanded to manage the status, power distribution, and charging plans of multiple charging points.
   - **User Access Control:** Adding user management features with different levels of access, for example, administrators can modify system settings, while regular users can only view the charging status or submit charging requests.

9 Acknowledgements

We appreciate Mr. Thomas Franzelin, M.Sc. for his invaluable support and assistance throughout this project. His guidance in upgrading the charging station from a conventional system to one capable of smart charging, thoughtful advice on hardware selection, and remarkable patience during our technical discussions were instrumental to the successful completion of this work. We are also sincerely thankful for his continuous support in providing timely access to the testing platform whenever needed, which played a key role in enabling the development and validation work throughout the project. His dedication, expertise, and willingness to help at every stage are deeply appreciated.

Bibliography

[1] *AF30-30-00-13 | ABB*. URL: https://new.abb.com/products/1SBL277001R1300/af30-30-00-13 (visited on 03/19/2025).

[2] International Electrotechnical Commission. *IEC 62196-2:2022*. International Electrotechnical Commission, 2022. ISBN: 9782832259313.

[3] *Configuration - Raspberry Pi Documentation*. URL: https://www.raspberrypi.com/documentation/computers/configuration.html#configure-uarts (visited on 03/24/2025).

[4] Adam Creavin. *A Guide to EV Charging Station Management Systems - Wayleadr*. URL: https://wayleadr.com/blog/ev-charging-station-management-systems-a-comprehensive-guide-for-workplaces/ (visited on 07/30/2024).

[5] *Electric Vehicles*. URL: https://www.statista.com/outlook/mmo/electric-vehicles/worldwide (visited on 01/2025).

[6] *evracing.cz dir list*. Resource: https://evracing.cz/evse/evse-wallbox/. URL: https://evracing.cz/evse/evse-wallbox/evse-wb-din_latest.pdf (visited on 03/19/2025).

[7] Neal Ford et al. *Software Architecture: The Hard Parts: Modern Trade-Off Analyses for Distributed Architectures*. O'Reilly Media, 2021. ISBN: 978-1-492-08689-5.

[8] Erich Gamma et al. *Design Patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 2009. ISBN: 0-201-63361-2.

[9] *Info & whitepapers - Open Charge Alliance*. URL: https://openchargealliance.org/whitepapers/ (visited on 03/21/2025).

[10] International Renewable Energy Agency(IRENA). *Innovation Outlook Smart Charging For Electric Vehicles*. Tech. rep. International Renewable Energy Agency(IRENA), 2019.

[11] mobilityhouse. *Github: mobilityhouse/ocpp*. URL: https://github.com/mobilityhouse/ocpp (visited on 03/24/2025).

[12] Modbus Organization. *Modbus Application Protocol V1.1b3*. Tech. rep. Modbus Organization, 2012.

[13] Scott Robinson. *Coroutine - Tutorial*. URL: https://stackabuse.com/python-async-await-tutorial/ (visited on 12/29/2015).

[14] *Shelly Pro 3EM – Shelly Europe*. URL: https://www.shelly.com/de/products/shelly-pro-3em-x1?srsltid=AfmBOopAsMzkel_HYApY229XHyU0HtRgCnETvK_sRnCDi0549gWNNiaw (visited on 03/15/2024).

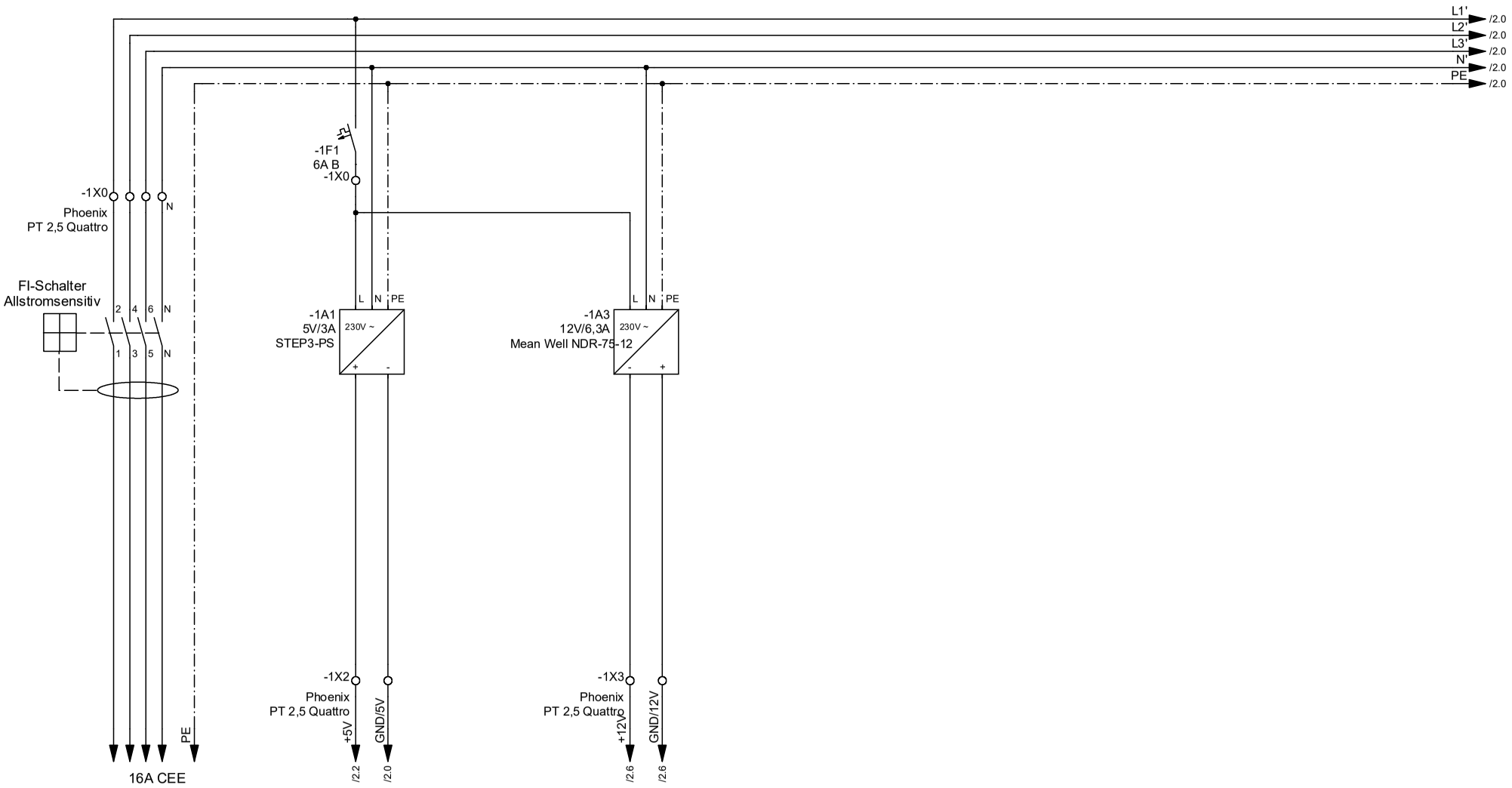
[15] Johan Thelin. *Foundations of Qt Development*. Apress, 2007. ISBN: 978-1-59059-831-3.

[16] *WebSocket - Standard*. URL: https://datatracker.ietf.org/doc/html/rfc6455 (visited on 12/2011).

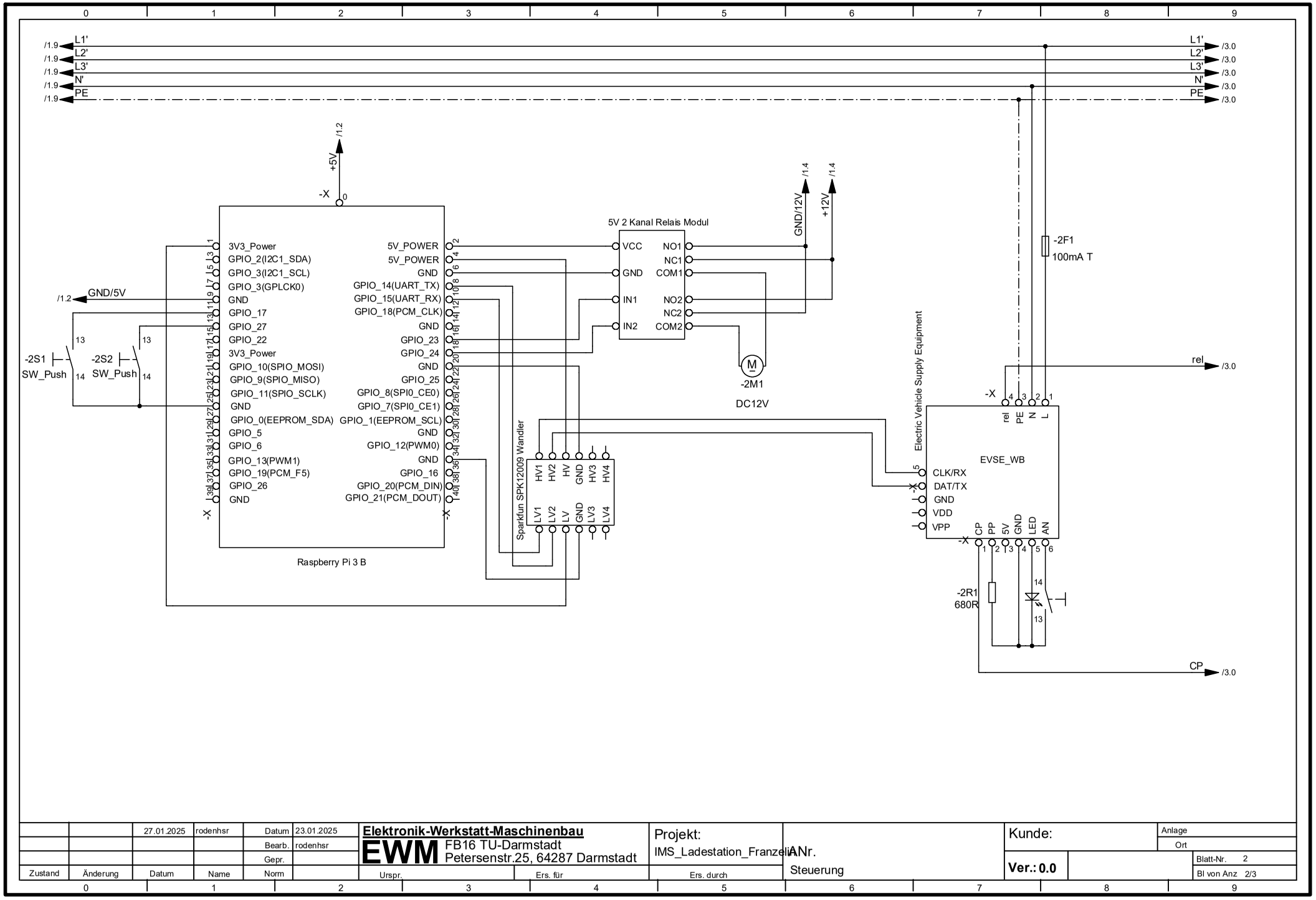
[17] Ryan Winters. *Raspberry Pi Overview*. URL: https://www.jameco.com/Jameco/workshop/CircuitNotes/Circuit_Notes_RasberryPiPDF.pdf (visited on 03/24/2024).

10 Appendices

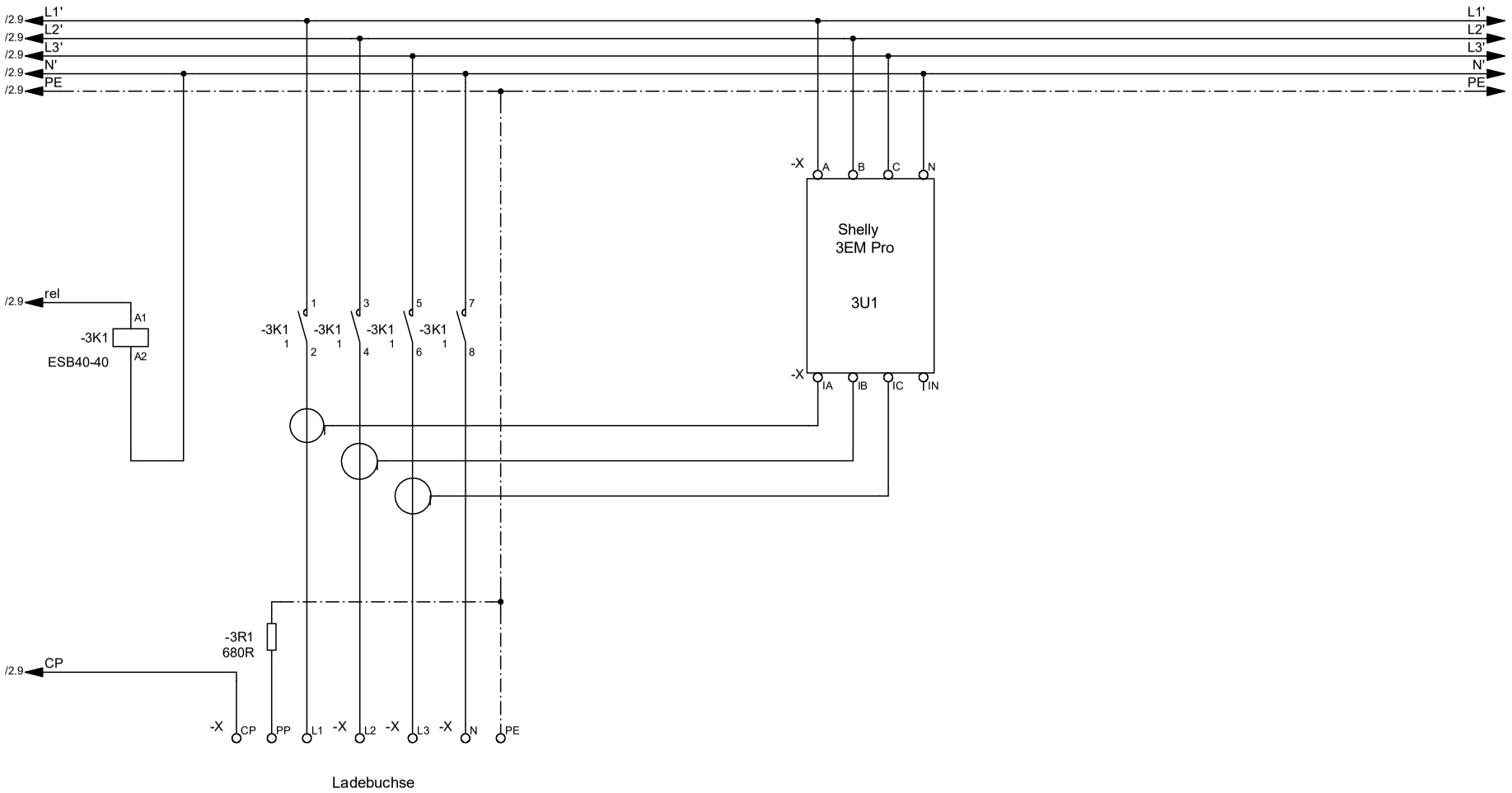
1. Ladestation_v2R
2. Use Case Diagram
3. Classes Diagram 1
4. Classes Diagram 2
5. Components Diagram
6. Flow Diagram
7. Simulation test cases



		29.01.2025	rodenhsr	Datum	23.01.2025	**Elektronik-Werkstatt-Maschinenbau**	Projekt:		Kunde:	Anlage
				Bearb.	rodenhsr	**EWM** FB16 TU-Darmstadt	IMS_Ladestation_Franzeli	ANr.		Ort
				Gepr.		Petersenstr.25, 64287 Darmstadt			**Ver.: 0.0**	Blatt-Nr. 1
Zustand	Änderung	Datum	Name	Norm		Urspr. / Ers. für	Ers. durch	Ladestation Einspeisung und Netzteile		Bl von Anz 1/3



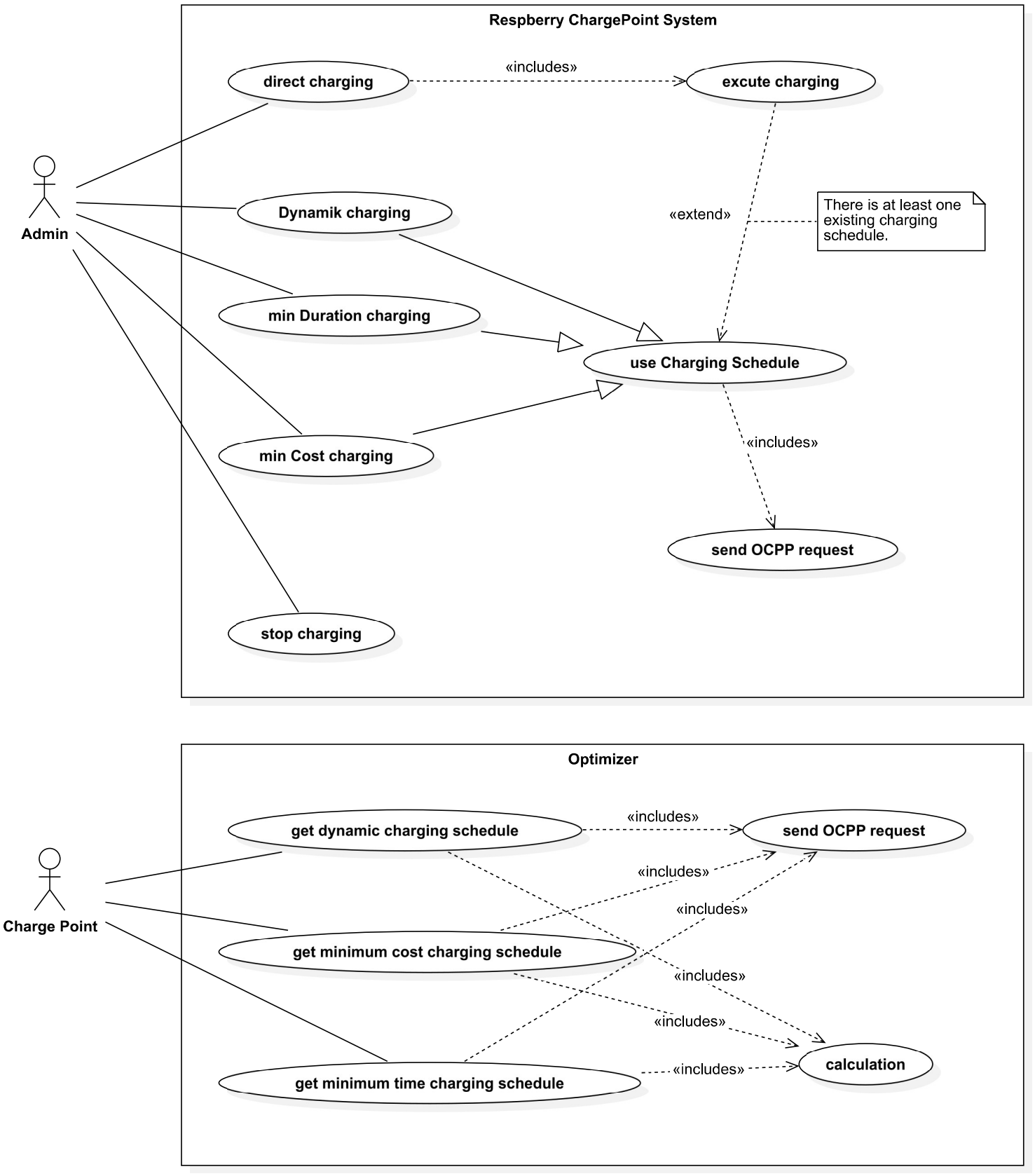
		27.01.2025	rodenhsr	Datum	23.01.2025	**Elektronik-Werkstatt-Maschinenbau**	Projekt:		Kunde:	Anlage	
				Bearb.	rodenhsr	**EWM** FB16 TU-Darmstadt	IMS_Ladestation_Franzeli	ANr.		Ort	
				Gepr.		Petersenstr.25, 64287 Darmstadt		Steuerung	**Ver.: 0.0**		Blatt-Nr. 2
Zustand	Änderung	Datum	Name	Norm		Urspr. / Ers. für	Ers. durch				Bl von Anz 2/3



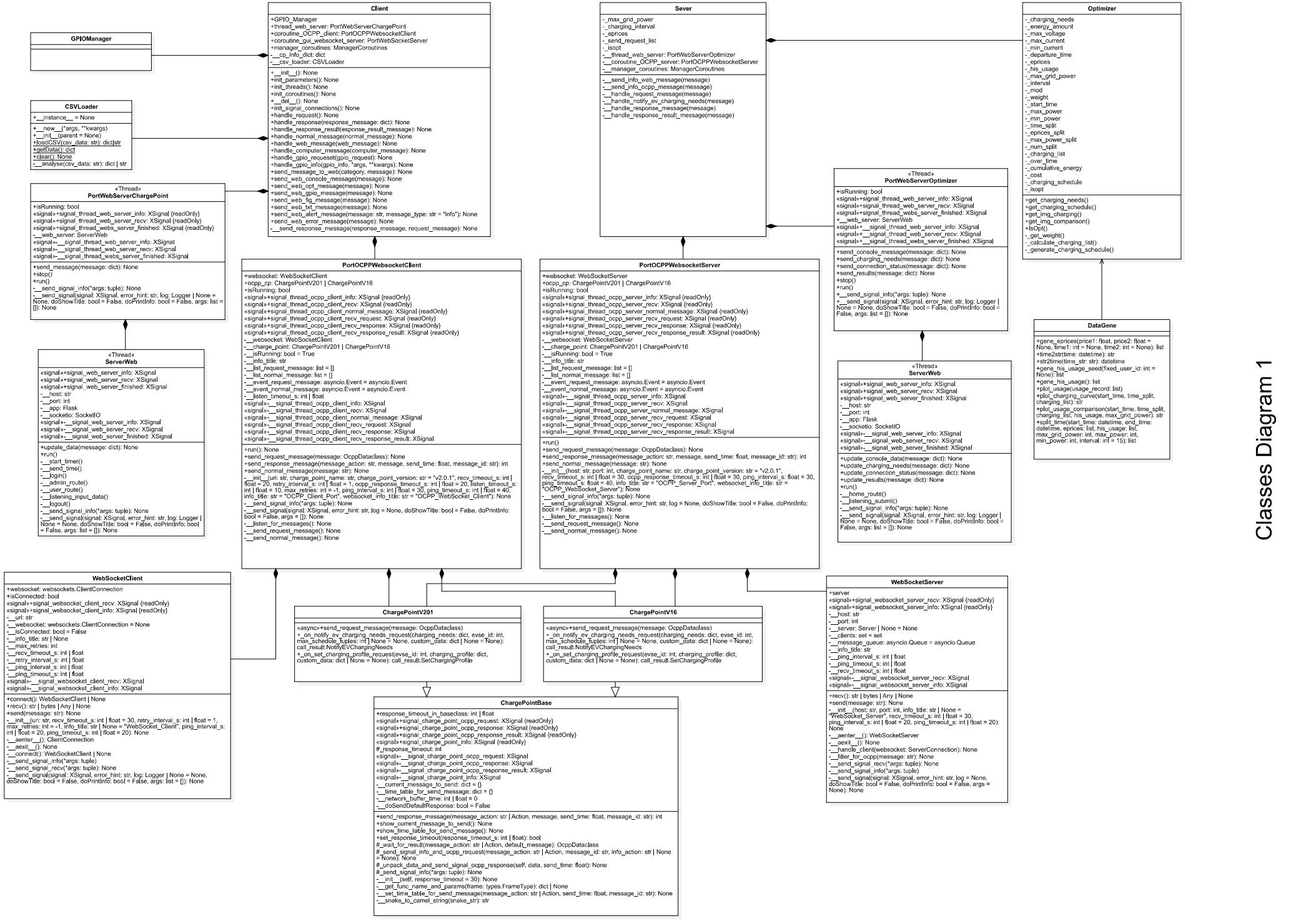
Ladebuchse

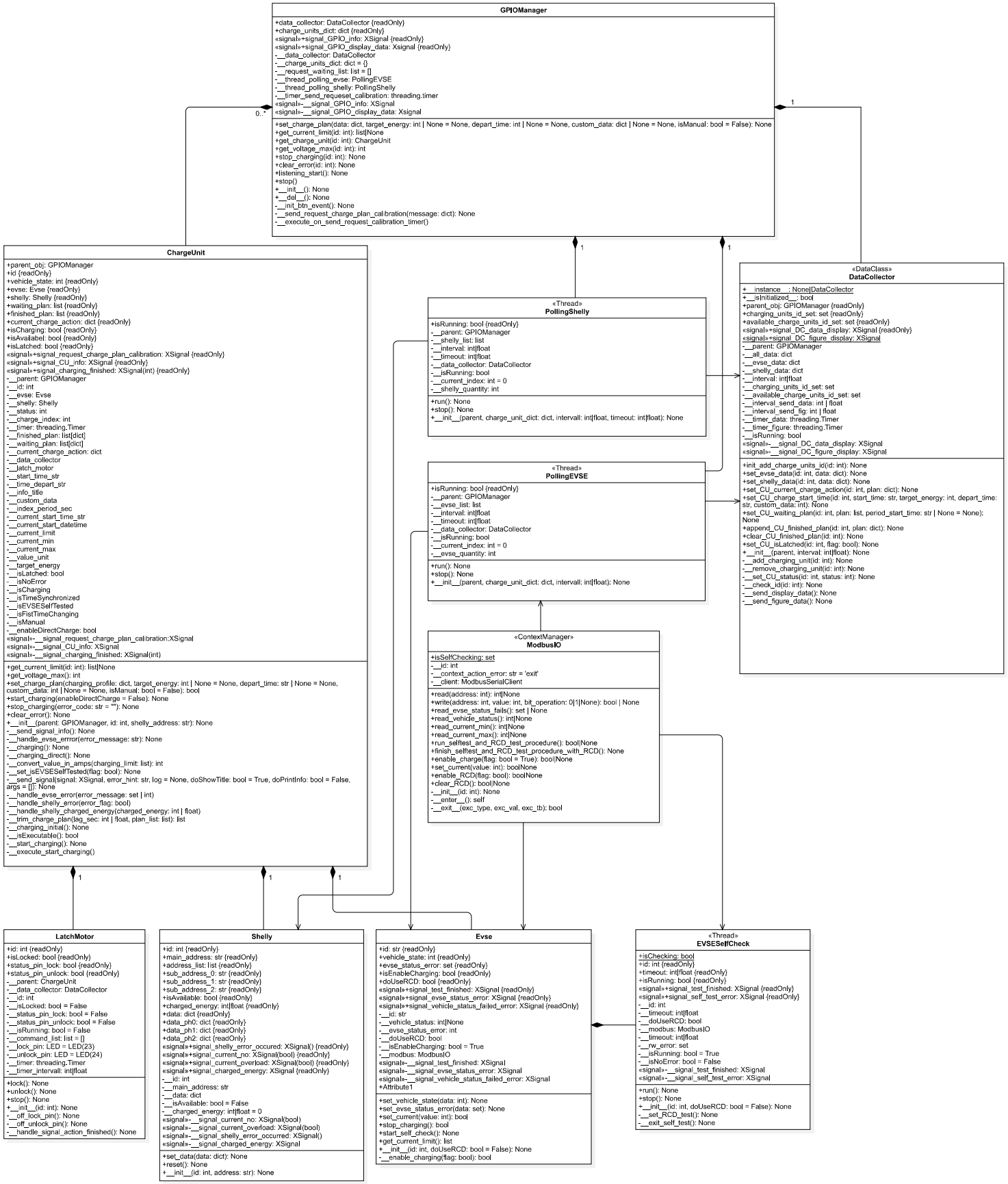
1 2 2
3 4 2
5 6 2
7 8 3

		27.01.2025	rodenhsr	Datum	24.01.2025	**Elektronik-Werkstatt-Maschinenbau**	Projekt:		Kunde:	Anlage	
				Bearb.	rodenhsr	**EWM** FB16 TU-Darmstadt	IMS_Ladestation_Franzeli	ANr.		Ort	
				Gepr.		Petersenstr.25, 64287 Darmstadt			**Ver.: 0.0**		Blatt-Nr. 3
Zustand	Änderung	Datum	Name	Norm		Urspr. / Ers. für	Ers. durch	Steuerung			Bl von Anz 3/3

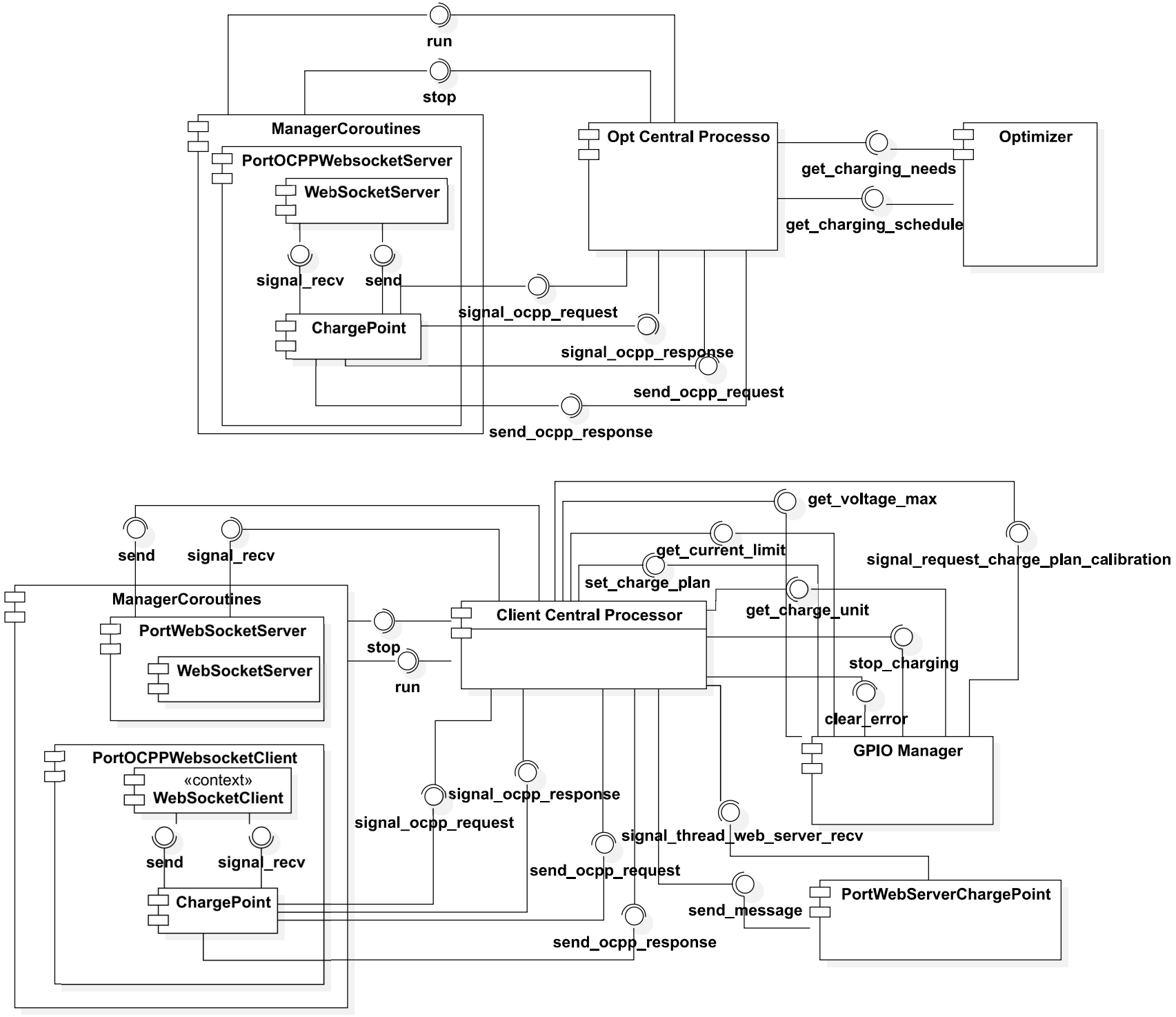


Use Case Diagram

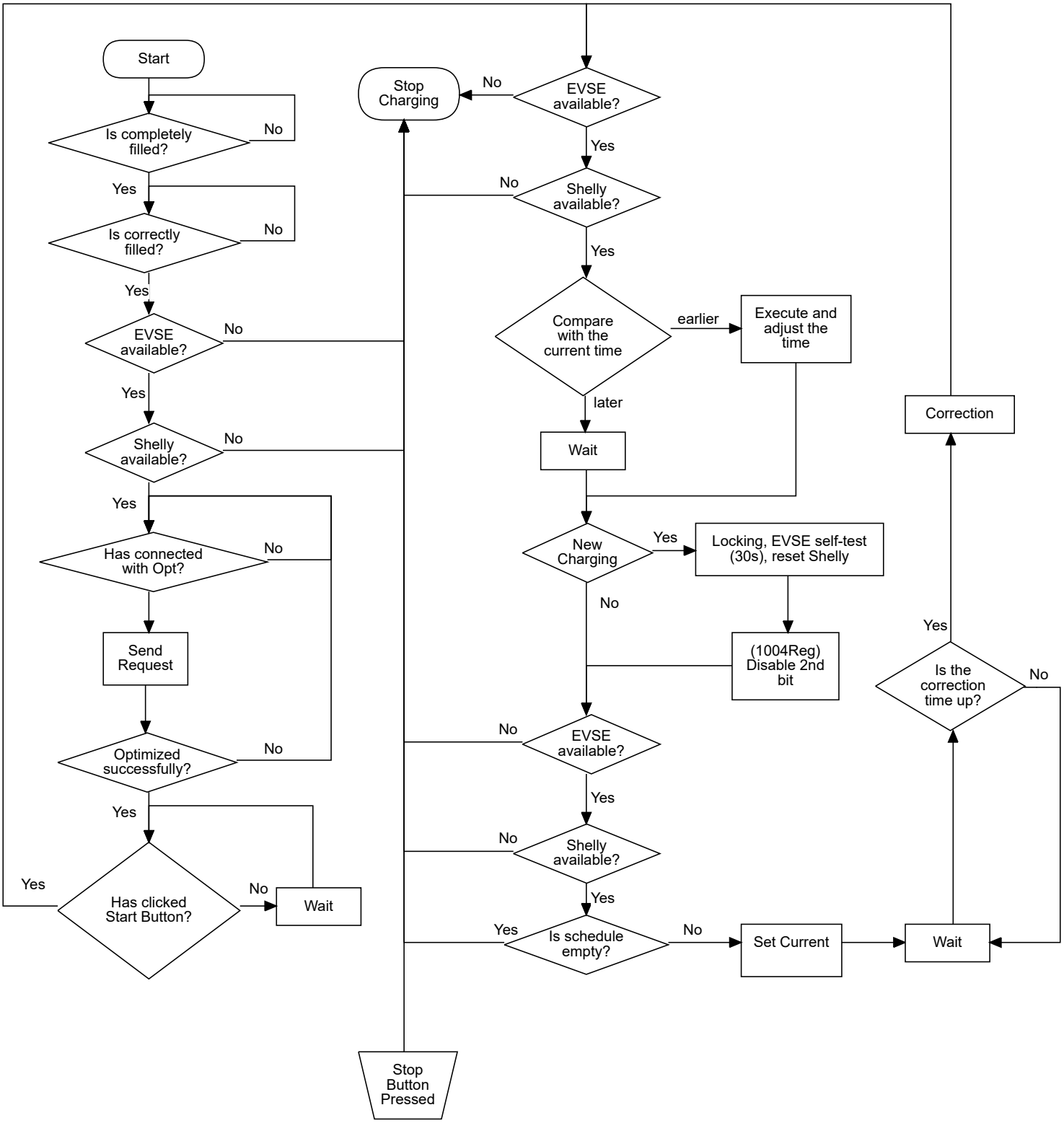

Client
+GPIO_Manager
+thread_web_server: PortWebServerChargePoint
+coroutine_OCPP_client: PortOCPPWebsocketClient
+coroutine_gui_websocket_server: PortWebSocketServer
+manager_coroutines: ManagerCoroutines
-__cp_info_dict: dict
-__csv_loader: CSVLoader
+__init__(): None
+init_parameters(): None
+init_threads(): None
+init_coroutines(): None
+__del__(): None
+init_signal_connections(): None
+handle_request(): None
+handle_response(response_message: dict): None
+handle_response_result(response_result_message): None
+handle_normal_message(normal_message): None
+handle_web_message(web_message): None
+handle_computer_message(computer_message): None
+handle_gpio_requeset(gpio_request): None
+handle_gpio_info(gpio_info, *args, **kwargs): None
+send_message_to_web(category, message): None
+send_web_console_message(message): None
+send_web_opt_message(message): None
+send_web_gpio_message(message): None
+send_web_fig_message(message): None
+send_web_txt_message(message): None
+send_web_alert_message(message: str, message_type: str = "info"): None
+send_web_error_message(message): None
-__send_response_message(response_message, request_message): None
GPIOManager
CSVLoader
+__instance__ = None
+__new__(*args, **kwargs)
+__init__(parent = None)
+loadCSV(csv_data: str): dict|str
+getData(): dict
+clear(): None
-__analyse(csv_data: str): dict | str
Sever
-_max_grid_power
-_charging_interval
-_eprices
-_send_request_list
-_isopt
-__thread_web_server: PortWebServerOptimizer
-__coroutine_OCPP_server: PortOCPPWebsocketServer
-__manager_coroutines: ManagerCoroutines
-__send_info_web_message(message)
-__send_info_ocpp_message(message)
-__handle_request_message(message)
-__handle_notify_ev_charging_needs(message)
-__handle_response_message(message)
-__handle_response_result_message(message)
Optimizer
-_charging_needs
-_energy_amount
-_max_voltage
-_max_current
-_min_current
-_departure_time
-_eprices
-_his_usage
-_max_grid_power
-_interval
-_mod
-_weight
-_start_time
-_max_power
-_min_power
-_time_split
-_eprices_split
-_max_power_split
-_num_split
-_charging_list
-_over_time
-_cumulative_energy
-_cost
-_charging_schedule
-_isopt
+get_charging_needs()
+get_charging_schedule()
+get_img_charging()
+get_img_comparison()
+IsOpt()
-_get_weight()
-_calculate_charging_list()
-_generate_charging_schedule()
«Thread»
PortWebServerChargePoint
+isRunning: bool
«signal»+signal_thread_web_server_info: XSignal {readOnly}
«signal»+signal_thread_web_server_recv: XSignal {readOnly}
«signal»+signal_thread_webs_server_finished: XSignal {readOnly}
-__web_server: ServerWeb
«signal»-__signal_thread_web_server_info: XSignal
«signal»-__signal_thread_web_server_recv: XSignal
«signal»-__signal_thread_webs_server_finished: XSignal
+send_message(message: dict): None
+stop()
+run()
-__send_signal_info(*args: tuple): None
-__send_signal(signal: XSignal, error_hint: str, log: Logger | None = None, doShowTitle: bool = False, doPrintInfo: bool = False, args: list = []): None
«Thread»
PortWebServerOptimizer
+isRunning: bool
«signal»+signal_thread_web_server_info: XSignal
«signal»+signal_thread_web_server_recv: XSignal
«signal»+signal_thread_webs_server_finished: XSignal
+__web_server: ServerWeb
«signal»+__signal_thread_web_server_info: XSignal
«signal»+__signal_thread_web_server_recv: XSignal
«signal»+__signal_thread_webs_server_finished: XSignal
+send_console_message(message: dict): None
+send_charging_needs(message: dict): None
+send_connection_status(message: dict): None
+send_results(message: dict): None
+stop()
+run()
+__send_signal_info(*args: tuple): None
+__send_signal(signal: XSignal, error_hint: str, log: Logger | None = None, doShowTitle: bool = False, doPrintInfo: bool = False, args: list = []): None
«Thresd»
ServerWeb
«signal»+signal_web_server_info: XSignal
«signal»+signal_web_server_recv: XSignal
«signal»+signal_web_server_finished: XSignal
-__host: str
-__port: int
-__app: Flask
-__socketio: SocketIO
«signal»-__signal_web_server_info: XSignal
«signal»-__signal_web_server_recv: XSignal
«signal»-__signal_web_server_finished: XSignal
+update_console_data(message: dict): None
+update_charging_needs(message: dict): None
+update_connection_status(message: dict): None
+update_results(message: dict): None
+run()
-__home_route()
-__listening_submit()
-__send_signal_info(*args: tuple): None
-__send_signal(signal: XSignal, error_hint: str, log: Logger | None = None, doShowTitle: bool = False, doPrintInfo: bool = False, args: list = []): None
DataGene
+gene_eprices(price1: float, price2: float = None, time1: int = None, time2: int = None): list
+time2str(time: datetime): str
+str2time(time_str: str): datetime
+gene_his_usage_seed(fixed_user_id: int = None): list
+gene_his_usage(): list
+plot_usage(usage_record: list)
+plot_charging_curve(start_time, time_split, charging_list): str
+plot_usage_comparison(start_time, time_split, charging_list, his_usage, max_grid_power): str
+split_time(start_time: datetime, end_time: datetime, eprices: list, his_usage: list, max_grid_power: int, max_power: int, min_power: int, interval: int = 15): list
«Thread»
ServerWeb
«signal»+signal_web_server_info: XSignal
«signal»+signal_web_server_recv: XSignal
«signal»+signal_web_server_finished: XSignal
-__host: str
-__port: int
-__app: Flask
-__socketio: SocketIO
«signal»-__signal_web_server_info: XSignal
«signal»-__signal_web_server_recv: XSignal
«signal»-__signal_web_server_finished: XSignal
+update_data(message: dict): None
+run()
-__start_timer()
-__send_time()
-__login()
-__admin_route()
-__user_route()
-__listening_input_data()
-__logout()
-__send_signal_info(*args: tuple): None
-__send_signal(signal: XSignal, error_hint: str, log: Logger | None = None, doShowTitle: bool = False, doPrintInfo: bool = False, args: list = []): None
PortOCPPWebsocketClient
+websocket: WebSocketClient
+ocpp_cp: ChargePointV201 | ChargePointV16
+isRunning: bool
«signal»+signal_thread_ocpp_client_info: XSignal {readOnly}
«signal»+signal_thread_ocpp_client_recv: XSignal {readOnly}
«signal»+signal_thread_ocpp_client_normal_message: XSignal {readOnly}
«signal»+signal_thread_ocpp_client_recv_request: XSignal {readOnly}
«signal»+signal_thread_ocpp_client_recv_response: XSignal {readOnly}
«signal»+signal_thread_ocpp_client_recv_response_result: XSignal {readOnly}
-__websocket: WebSocketClient
-__charge_point: ChargePointV201 | ChargePointV16
-__isRunning: bool = True
-__info_title: str
-__list_request_message: list = []
-__list_normal_message: list = []
-__event_request_message: asyncio.Event = asyncio.Event
-__event_normal_message: asyncio.Event = asyncio.Event
-__listen_timeout_s: int | float
«signal»-__signal_thread_ocpp_client_info: XSignal
«signal»-__signal_thread_ocpp_client_recv: XSignal
«signal»-__signal_thread_ocpp_client_normal_message: XSignal
«signal»-__signal_thread_ocpp_client_recv_request: XSignal
«signal»-__signal_thread_ocpp_client_recv_response: XSignal
«signal»-__signal_thread_ocpp_client_recv_response_result: XSignal
+run(): None
+send_request_message(message: OcppDataclass): None
+send_response_message(message_action: str, message, send_time: float, message_id: str): int
+send_normal_message(message: str): None
-__init__(uri: str, charge_point_name: str, charge_point_version: str = "v2.0.1", recv_timeout_s: int | float = 20, retry_interval_s: int | float = 1, ocpp_response_timeout_s: int | float = 20, listen_timeout_s: int | float = 10, max_retries: int = -1, ping_interval_s: int | float = 30, ping_timeout_s: int | float = 40, info_title: str = "OCPP_Client_Port", websocket_info_title: str = "OCPP_WebSocket_Client"): None
-__send_signal_info(*args: tuple): None
-__send_signal(signal: XSignal, error_hint: str, log = None, doShowTitle: bool = False, doPrintInfo: bool = False, args = []): None
-__listen_for_messages(): None
-__send_request_message(): None
-__send_normal_message(): None
PortOCPPWebsocketServer
+websocket: WebSocketServer
+ocpp_cp: ChargePointV201 | ChargePointV16
+isRunning: bool
«signal»+signal_thread_ocpp_server_info: XSignal {readOnly}
«signal»+signal_thread_ocpp_server_recv: XSignal {readOnly}
«signal»+signal_thread_ocpp_server_normal_message: XSignal {readOnly}
«signal»+signal_thread_ocpp_server_recv_request: XSignal {readOnly}
«signal»+signal_thread_ocpp_server_recv_response: XSignal {readOnly}
«signal»+signal_thread_ocpp_server_recv_response_result: XSignal {readOnly}
-__websocket: WebSocketServer
-__charge_point: ChargePointV201 | ChargePointV16
-__isRunning: bool = True
-__info_title: str
-__list_request_message: list = []
-__list_normal_message: list = []
-__event_request_message: asyncio.Event = asyncio.Event
-__event_normal_message: asyncio.Event = asyncio.Event
«signal»-__signal_thread_ocpp_server_info: XSignal
«signal»-__signal_thread_ocpp_server_recv: XSignal
«signal»-__signal_thread_ocpp_server_normal_message: XSignal
«signal»-__signal_thread_ocpp_server_recv_request: XSignal
«signal»-__signal_thread_ocpp_server_recv_response: XSignal
«signal»-__signal_thread_ocpp_server_recv_response_result: XSignal
+run()
+send_request_message(message: OcppDataclass): None
+send_response_message(message_action: str, message, send_time: float, message_id: str): int
+send_normal_message(message: str): None
-__init__(host: str, port: int, charge_point_name: str, charge_point_version: str = "v2.0.1", recv_timeout_s: int | float = 30, ocpp_response_timeout_s: int | float = 30, ping_interval_s: float = 30, ping_timeout_s: float = 40, info_title: str = "OCPP_Server_Port", websocket_info_title: str = "OCPP_WebSocket_Server"): None
-__send_signal_info(*args: tuple): None
-__send_signal(signal: XSignal, error_hint: str, log = None, doShowTitle: bool = False, doPrintInfo: bool = False, args = []): None
-__listen_for_messages(): None
-__send_request_message(): None
-__send_normal_message(): None
WebSocketClient
+websocket: websockets.ClientConnection
+isConnected: bool
«signal»+signal_websocket_client_recv: XSignal {readOnly}
«signal»+signal_websocket_client_info: XSignal {readOnly}
-__uri: str
-__websocket: websockets.ClientConnection = None
-__isConnected: bool = False
-__info_title: str | None
-__max_retries: int
-__recv_timeout_s: int | float
-__retry_interval_s: int | float
-__ping_interval_s: int | float
-__ping_timeout_s: int | float
«signal»-__signal_websocket_client_recv: XSignal
«signal»-__signal_websocket_client_info: XSignal
+connect(): WebSocketClient | None
+recv(): str | bytes | Any | None
+send(message: str): None
-__init__(uri: str, recv_timeout_s: int | float = 30, retry_interval_s: int | float = 1, max_retries: int = -1, info_title: str | None = "WebSocket_Client", ping_interval_s: int | float = 20, ping_timeout_s: int | float = 20): None
-__aenter__(): ClientConnection
-__aexit__(): None
-__connect(): WebSocketClient | None
-__send_signal_info(*args: tuple)
-__send_signal_recv(*args: tuple): None
-__send_signal(signal: XSignal, error_hint: str, log: Logger | None = None, doShowTitle: bool = False, doPrintInfo: bool = False, args: list = []): None
ChargePointV201
«async»+send_request_message(message: OcppDataclass)
+_on_notify_ev_charging_needs_request(charging_needs: dict, evse_id: int, max_schedule_tuples: int | None = None, custom_data: dict | None = None): call_result.NotifyEVChargingNeeds
+_on_set_charging_profile_request(evse_id: int, charging_profile: dict, custom_data: dict | None = None): call_result.SetChargingProfile
ChargePointV16
«async»+send_request_message(message: OcppDataclass)
+_on_notify_ev_charging_needs_request(charging_needs: dict, evse_id: int, max_schedule_tuples: int| None = None, custom_data: dict | None = None): call_result.NotifyEVChargingNeeds
+_on_set_charging_profile_request(evse_id: int, charging_profile: dict, custom_data: dict | None = None): call_result.SetChargingProfile
WebSocketServer
+server
«signal»+signal_websocket_server_recv: XSignal {readOnly}
«signal»+signal_websocket_server_info: XSignal {readOnly}
-__host: str
-__port: int
-__server: Server | None = None
-__clients: set = set
-__message_queue: asyncio.Queue = asyncio.Queue
-__info_title: str
-__ping_interval_s: int | float
-__ping_timeout_s: int | float
-__recv_timeout_s: int | float
«signal»-__signal_websocket_server_recv: XSignal
«signal»-__signal_websocket_server_info: XSignal
+recv(): str | bytes | Any | None
+send(message: str): None
-__init__(host: str, port: int, info_title: str | None = "WebSocket_Server", recv_timeout_s: int | float = 30, ping_interval_s: int | float = 20, ping_timeout_s: int | float = 20): None
-__aenter__(): WebSocketServer
-__aexit__(): None
-__handle_client(websocket: ServerConnection): None
-__filter_for_ocpp(message: str): None
-__send_signal_recv(*args: tuple): None
-__send_signal_info(*args: tuple)
-__send_signal(signal: XSignal, error_hint: str, log = None, doShowTitle: bool = False, doPrintInfo: bool = False, args = None): None
ChargePointBase
+response_timeout_in_baseclass: int | float
«signal»+signal_charge_point_ocpp_request: XSignal {readOnly}
«signal»+signal_charge_point_ocpp_response: XSignal {readOnly}
«signal»+signal_charge_point_ocpp_response_result: XSignal {readOnly}
«signal»+signal_charge_point_info: XSignal {readOnly}
#_response_timeout: int
«signal»-__signal_charge_point_ocpp_request: XSignal
«signal»-__signal_charge_point_ocpp_response: XSignal
«signal»-__signal_charge_point_ocpp_response_result: XSignal
«signal»-__signal_charge_point_info: XSignal
-__current_message_to_send: dict = {}
-__time_table_for_send_message: dict = {}
-__network_buffer_time: int | float = 0
-__doSendDefaultResponse: bool = False
+send_response_message(message_action: str | Action, message, send_time: float, message_id: str): int
+show_current_message_to_send(): None
+show_time_table_for_send_message(): None
+set_response_timeout(response_timeout_s: int | float): bool
#_wait_for_result(message_action: str | Action, default_message): OcppDataclass
#_send_signal_info_and_ocpp_request(message_action: str | Action, message_id: str, info_action: str | None = None): None
#_unpack_data_and_send_signal_ocpp_response(self, data, send_time: float): None
#_send_signal_info(*args: tuple): None
-__init__(self, response_timeout = 30): None
-__get_func_name_and_params(frame: types.FrameType): dict | None
-__set_time_table_for_send_message(message_action: str | Action, send_time: float, message_id: str): None
-__snake_to_camel_string(snake_str): str
Classes Diagram 1



Classes Diagram 2



Component Diagram



Flow Diagram

Simulation test cases				
Phase	Module	Description	Expectation	result
Before charging	Web	Incorrect input	Show hint, halt step	As expected
		incompletely filled out	Show hint, halt step	As expected
		EVSE in use/ Not exist/Error	Halt step	As expected
		Shelly not available	Halt step	As expected
		Correctly filled in, device ready	Send request to the optimizer	As expected
		click Start directly	No Reaction	As expected
		click Stop directly	No Reaction	As expected
	Optimizer	Disconnect with the optimizer	Wait for the connection	As expected
		Connect with the Opt. Again	Connect again	As expected
		Error in the optimizer	Halt step	As expected
		Succeeded	Charge-Plan receive	As expected
	Vehicle	No car plugged in	Halt step	As expected
		Car plugged in	Charging ready	As expected
	Button (Web)	Click Start without filling in	No Reaction	As expected
		Click Stop without filling in	No Reaction	As expected
		Click Reset-Error without Error	No Reaction	As expected
		Status error, click Start	No Reaction	As expected
		Status error, click Stop	No Reaction	As expected
		Status error, Click Reset-Error	Error reset, contact the administrator	As expected
		Steps finished, status normal	Charging ready	As expected
Beginning of charging	EVSE	No car plugged in	Not allowed to start charging	As expected
		Error	Not allowed to start charging	As expected
		Status available	Charging ready	As expected
	Shelly	Error	Not allowed to start charging	As expected
		Status available	Charging ready	As expected
	Start	Has not clicked start button	Wait, initialize	As expected
		Start clicked	Adjust schedule, lock, EVSE self-test	As expected
	EVSE Selftest	Succeed in self-test	Charging ready, wait	As expected
		Self-test failed	Error, Halt step	As expected
In charging	EVSE	Error	Stop Charging	As expected
	Shelly	Error	Stop Charging	As expected
	User	Click stop button	Stop Charging	As expected
		Do not stop, resend request	Request rejected	As expected
		Stop, Send request again	Start from the top	As expected
		start again	Continue executing, no effect	As expected
	Opt.	Disconnect with the optimizer	Continue executing, no effect	As expected
		Connected with the optimizer again	Continue executing, no effect	As expected
	Calibration	Connected with the optimizer	Periodic calibration	As expected
		Disconnected with the optimizer	Continue executing, no effect	As expected
Finish charging		Schedule list is empty	Stop Charging	As expected
		Energy fully charged	Stop Charging	As expected