Fachhochschule Nordwestschweiz
Hochschule für Technik

OpenCEM

Documentation (english)

Windisch, 23.08.2023

by Sergio Ferreira

Contents

List of Tables **4**

List of Figures **4**

1 OpenCEM Structure **6**

1.1 OpenCEM Class Structure 7

1.1.1 Communication Channels 8

1.1.2 Sensors 8

1.1.3 Actuators 9

1.1.4 Controllers 10

1.1.5 Devices 15

1.2 supported devices, sensors and actuators 19

1.3 OpenCEM Main 19

1.4 OpenCEM GUI 21

1.5 Simulation Mode 23

2 OpenCEM Configuration **25**

2.1 YAML Configuration File 25

2.2 YAML OpenCEM Settings 28

2.3 SmartGridready Devices 30

2.4 OpenCEM Web Configurator 31

3 Logging **34**

3.1 Device Statistics Logger 34

3.2 Creating Graphs 34

4 Commissioning **35**

4.1 Factory Reset 35

4.2 Verbindung und VNC 35

4.3 Python Update 38

4.4 GitHub on RevPi 38

4.5 OpenCEM Installation 39

4.6 OpenCEM Autostart 40

4.7 Test Setup 40

5 Error Handling **43**

6 OpenCEM Testing **45**

6.1 Durability Test 45

Quellenverzeichnis **48**

List of Tables

1.1 Terminal combinations of PICO charging station 9
1.2 Key Modbus registers of the used charging stations [3] 17
1.3 Existing endpoints of the OpenCEM GUI web server 23
4.1 Network addresses of the devices 42
5.1 Potential error cases to be handled. 43

List of Figures

1.1 Components of OpenCEM with indications of when they were implemented. . . . 6
1.2 Overview of the OpenCEM structure. 7
1.3 Overview of communication channels available in OpenCEM. 8
1.4 Sensor class structure in OpenCEM 8
1.5 Actuator class structure 9
1.6 Class structure of controllers in OpenCEM 10
1.7 Two household devices with *ExcessController* in a simulated OpenCEM system . 11
1.8 Heat pump with *StepwiseExcessController* in a simulated OpenCEM system . . . 12
1.9 EV charging station with *DynamicExcessController* in a simulated OpenCEM system 12
1.10 Device with *CoverageController* in a simulated OpenCEM system 13
1.11 Device with an *ExcessController* and a heat pump with a *PriceController* in a simulated OpenCEM system 14
1.12 Device class structure in OpenCEM 15
1.13 Vehicle status according to IEC 61851-1 [2] [**??**] 16
1.14 Charging process flowchart in OpenCEM 17
1.15 Implemented and tested devices, sensors, and actuators in OpenCEM (as of August 18, 2023). 19
1.16 Functioning of the *OpenCEM_mainV2.py* script 20
1.17 OpenCEM GUI of the test installation in a desktop browser 22
1.18 Simulation of the test installation 24
1.19 Supply and demand price of the price controller from Fig. 1.18 24
2.1 Basic structure of the YAML configuration file (Screenshot from PyCharm) . . . 25
2.2 UUID referencing and general and device-specific information. 26
2.3 Example YAML for a heat pump with multiple operating modes. 27

2.4 Example of a *StepwiseExcessController*. 27
2.5 YAML for a simulated heat pump. 28
2.6 Example of OpenCEM_settings.yaml 28
2.7 Example of an SGr file specified using a local path. 30
2.8 Example of an SGr file specified using a UUID. 30
2.9 Overview of an installation in the Web Configurator 31
2.10 Configuring a Power Meter 31
2.11 Interfaces between CEM Cloud and OpenCEM 32
2.12 Configuration of a Power Meter using an SGr File 33
4.1 Supported images for RevPi devices [7] 35
4.2 Network IP scan 36
4.3 Connecting to RevPi Connect+ using PuTTY. 36
4.4 Installing RealVNC on the RevPi device. 37
4.5 Open Raspberry Pi Config 37
4.6 Activating VNC in Raspberry Pi Config 37
4.7 Raspberry Pi reboot command 37
4.8 Enabling remote access using VNC Server on the RevPi Connect+. 38
4.9 Schematic of the test installation 40
4.10 Real installation with marked used devices. 41
4.11 Test setup with marked used devices. 42
5.1 USB-RS485 adapter disconnected from RevPi. 43
5.2 Wi-Fi temporarily turned off. 44
5.3 Incorrect login credentials entered for CEM Cloud. 44
6.1 Data of the installation on a day with changing cloud cover. 46
6.2 Temporal variation of production price and consumption price of the price controller of the electric heater 46
6.3 Data from August 16, 2023, illustrating the electric vehicle's charging process. . . 47

1 OpenCEM Structure

In Figure 1.1, the different components of OpenCEM are illustrated. Additionally, it indicates in which project these areas were implemented.

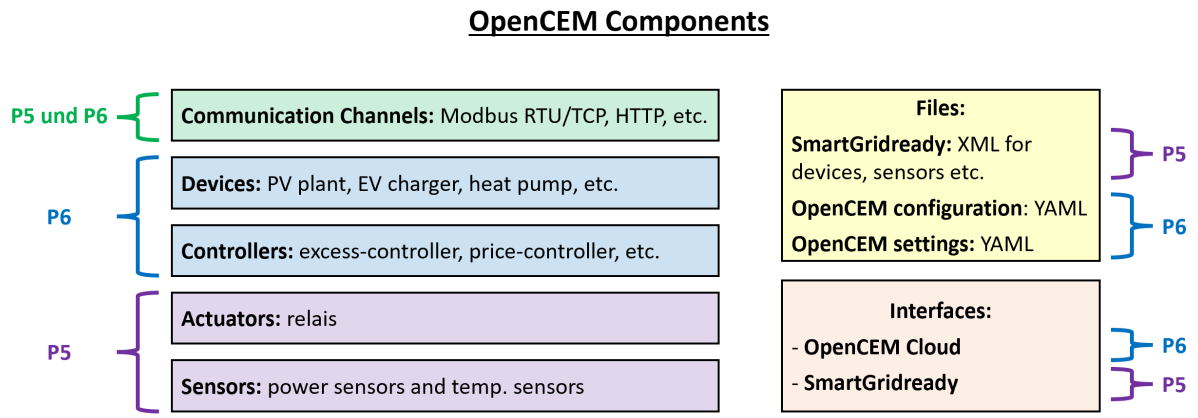

Figure 1.1: Components of OpenCEM with indications of when they were implemented.

OpenCEM is designed to be a modular and extensible self-consumption manager, which is why a well-thought-out class structure is at its core. It should provide a clear overview and be easily expandable. This chapter focuses on the structural organization of OpenCEM. It presents the existing classes and their relationships to each other.

Section 1.3 introduces the functionality of the OpenCEM Main Loop. To help users visualize what happens within OpenCEM, a graphical interface has been developed that displays essential information about connected devices. This is discussed in Chapter 1.4.

1.1 OpenCEM Class Structure

Figure 1.2 illustrates the class structure of the parent classes within OpenCEM. The class names were directly taken from the OpenCEM software, which is why they appear in English and CamelCase in the diagram. Details about the subclasses will be discussed in the subsequent chapters (1.1.1 - 1.1.5).

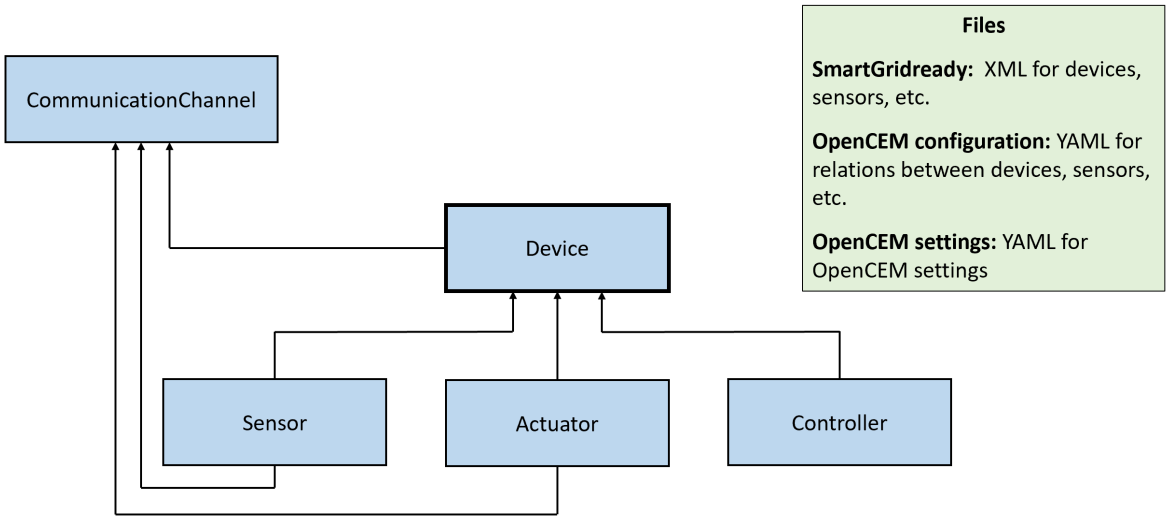

Figure 1.2: Overview of the OpenCEM structure.

The OpenCEM is structured with devices at its core. These devices can be associated with sensors, actuators, and controllers. To obtain information from sensors or control actuators, communication channels are required. These channels are consolidated using the *CommunicationChannel* class (e.g., Modbus RTU). As there are also intelligent devices that do not require external sensors or actuators, it is possible to communicate directly through the device. However, direct communication for devices has been implemented only for the existing EV charging station (Chapter 1.1.5). It is, however, intended to extend this to other devices (e.g., PV inverters) in the future.

For devices, sensors, or actuators that are supported by SmartGridready, information can be read in through SGr-XML description files. In order to make OpenCEM adaptable for different installations, a schema for YAML configuration files has been developed. Through these files, OpenCEM can be initialized (Chapter 2.1). Furthermore, settings can be adjusted via a YAML file (Chapter 2.2).

1.1.1 Communication Channels

The *CommunicationChannel* class consolidates various ways of communicating with a device. Necessary communication information is stored within instances of this class. Figure 1.3 presents the available communication channels in OpenCEM.

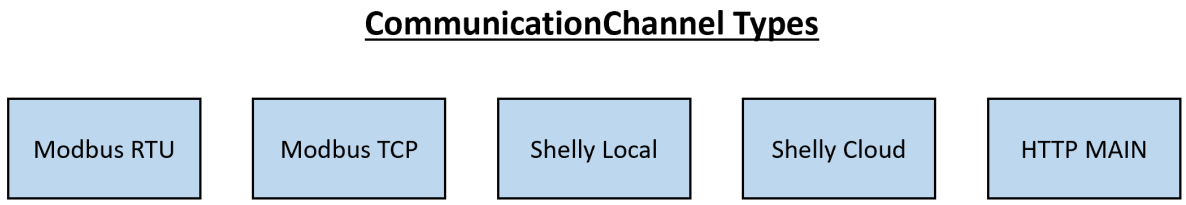

Figure 1.3: Overview of communication channels available in OpenCEM.

Devices, sensors, or actuators can be associated with communication channels through which communication takes place. A single channel can be used for multiple devices. In the case of Modbus RTU, it is even essential to use the same communication channel when devices are connected through the same hardware port. Failure to do so can result in an incorrect connection.

Communication via Modbus RTU, Shelly LOCAL, and SHELLY CLOUD was implemented in the previous semester. In P6, communication via Modbus TCP was added, which is used, for instance, in the electric vehicle charging station. Additionally, the *HTTP MAIN* communication channel was introduced for general communication with the internet or the GUI.

The SmartGridready library utilizes asynchronous functions from Python libraries *pyModbus* and *aiohttp*. To maintain uniformity, these libraries are also used for OpenCEM. The software developed in P5 was not asynchronous, requiring adjustments in this semester.

1.1.2 Sensors

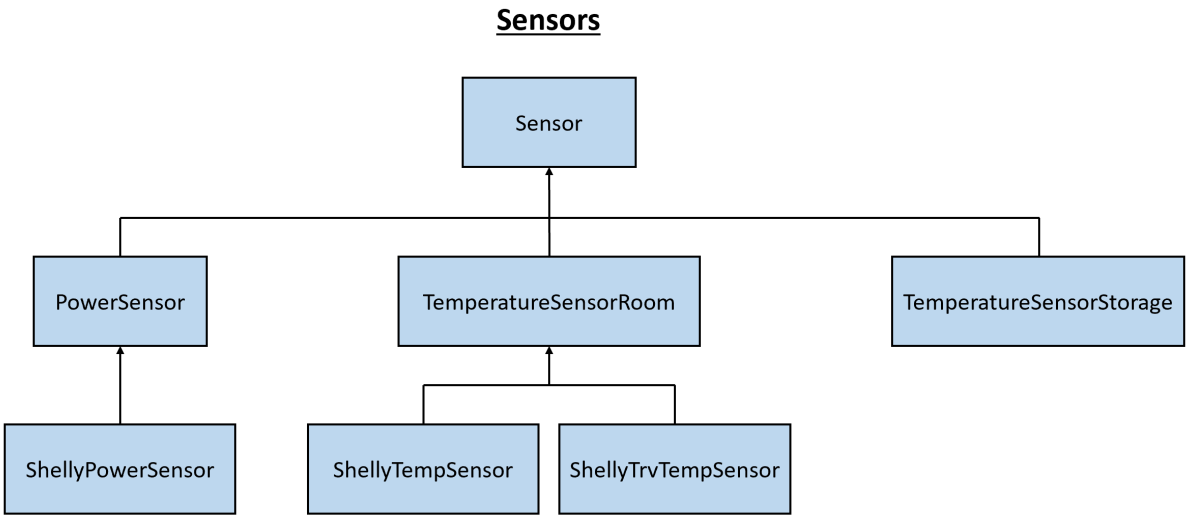

Figure 1.4: Sensor class structure in OpenCEM

Figure 1.4 illustrates the OpenCEM class structure of sensors. In this semester, only the *ShellyTrvTempSensor* has been added as a sensor for measuring room temperature. The remaining sensors were already implemented in P5 and are described in detail in the P5 project report.[1]

The Shelly TRV is an intelligent thermostat valve. The integrated temperature sensor is used in OpenCEM to read the room temperature. The device can be accessed through the *Shelly Local* or *Shelly Cloud* communication channels.

For the *TemperatureSensorStorage* class, no sensor was available for this project, which is why it has not been fully programmed yet.

1.1.3 Actuators

Figure 1.5 displays the class structure of actuators in OpenCEM. It's evident that there is not yet a wide selection of devices, making the structure quite simple. In P5, relays from the manufacturer Shelly were implemented. The tested devices included the *Pro 2PM* and *Pro 4PM*.

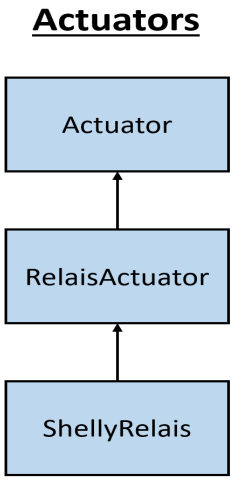

Figure 1.5: Actuator class structure

Adjustable Terminal Configurations

Devices with various operating modes often can be controlled through external terminal inputs. An example of this is the *PICO charging station*, which already supports SmartGridready. This charging station has two inputs that can adjust the charging power. Table 1.1 illustrates the terminal combinations and the resulting operating modes of the PICO charging station.

Input 1	Input 2	State
OFF	OFF	Maximum charging power
ON	OFF	Reduced charging (50% of max. charging power)
OFF	ON	Minimum charging power (6A)
ON	ON	No charging (0A)

Table 1.1: Terminal combinations of PICO charging station

Since these combinations can vary from device to device, a mechanism has been developed in OpenCEM to assign terminal combinations to operating modes. This makes it possible to control any devices with various power levels using a *StepwiseExcessController* (Chapter 1.1.4). The OpenCEM then switches the channels in a relay to achieve the desired level.

The configurations of the operating modes are stored in the YAML configuration file (Chapter 2.1).

1.1.4 Controllers

To efficiently utilize the generated surplus, the devices in the system need to be controlled. OpenCEM provides the following controllers as shown in Figure 1.6.

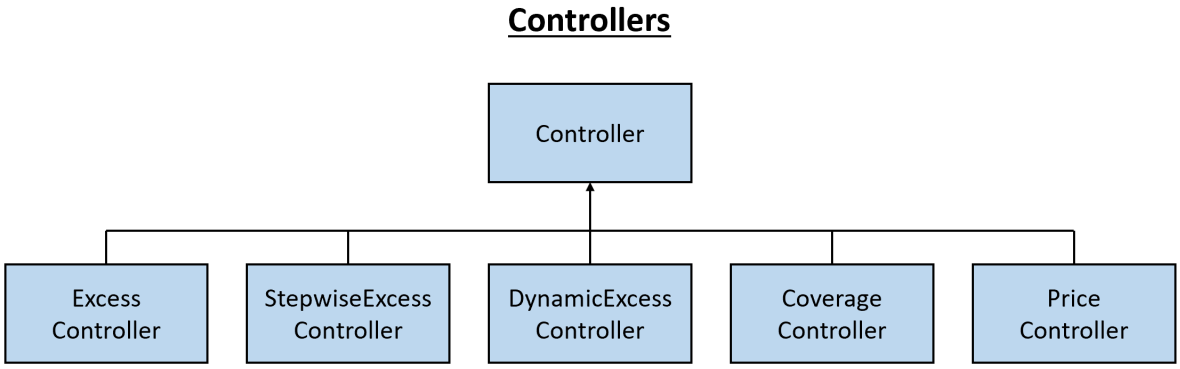

Figure 1.6: Class structure of controllers in OpenCEM

At the beginning of the project, the client provided the controllers as templates. However, they had not been tested previously. To implement these controllers in OpenCEM, some minor changes were made and errors in the code were fixed, as they were leading to incorrect behavior of the controllers. Additionally, in P6, the *DynamicExcessController* was added, which can be used for devices with dynamically adjustable power levels. The following section explains the available controllers and their functions, illustrated with simulated systems (also see Chapter 1.5 *Simulation Mode*).

Excess Controller

An excess controller calculates the state of a consumer device based on the surplus power in the system. The consumer device is the device that needs to be controlled. The surplus power is the power that is not currently needed by the overall system (usually generated by a PV system). An excess limit can be set for this controller, at which point the consumer device should be turned on.

To prevent all devices from turning on simultaneously when excess power is available, the power of the consumer devices is reserved. When a device is turned on, the nominal consumption is subtracted from the surplus. This mechanism is employed by all controllers present in OpenCEM.

Figure 1.7 shows two consumer devices each controlled by an excess controller. The system was simulated in OpenCEM for a 24-hour period. A simulated remaining consumption was added to the simulation, as this would also be present in reality due to unmeasured devices. The power produced by the photovoltaic system is simulated as a simple positive sinusoidal wave. The total system consumption is depicted in purple.

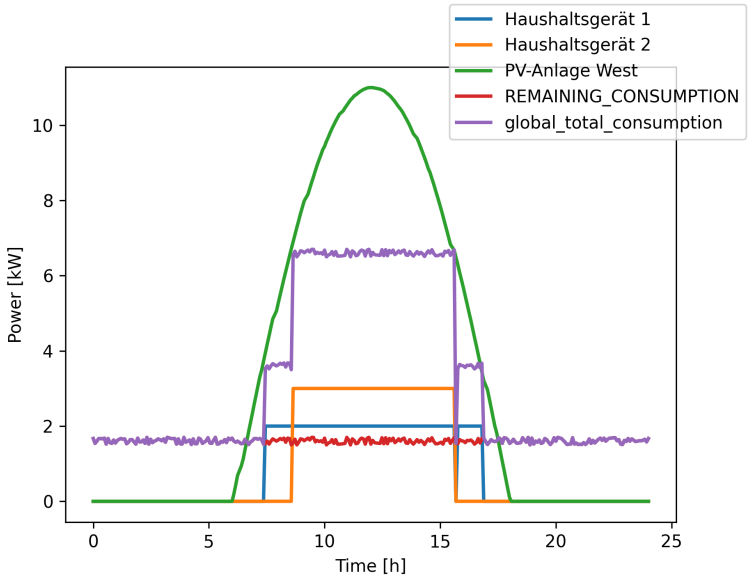

Figure 1.7: Two household devices with *ExcessController* in a simulated OpenCEM system

Stepwise Excess Controller

The stepwise excess controller differs from the regular excess controller in that it can switch between different levels. A list of excess limits can be specified. This controller is intended for devices that have multiple power levels, often seen in heat pumps or charging stations. Figure 1.8 depicts a simulated heat pump with multiple power levels controlled by a *StepwiseExcessController*.

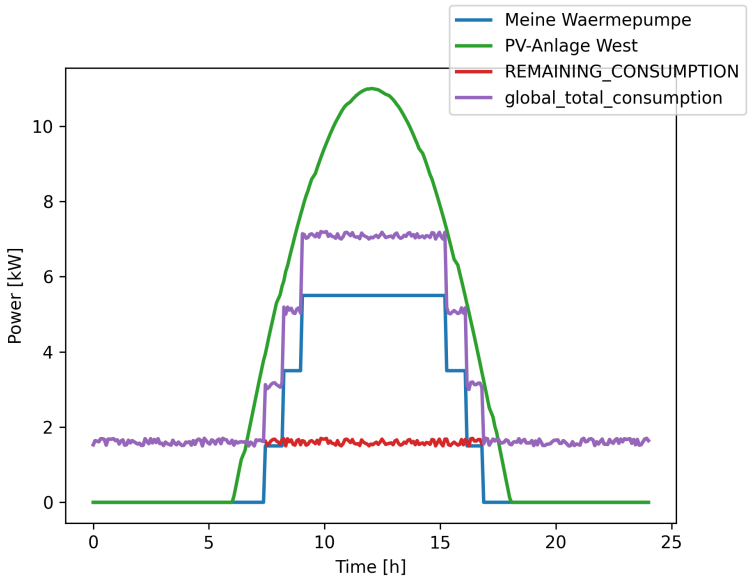

Figure 1.8: Heat pump with *StepwiseExcessController* in a simulated OpenCEM system

Dynamic Excess Controller

The dynamic excess controller differs from the regular excess controller in that it can continuously adjust the power consumption of a device. Naturally, the consumer device must also be capable of dynamically adjusting its consumption.

Figure 1.9 illustrates the functioning of the *DynamicExcessController* using a simulated example. This controller has upper and lower power limits that can be set. Between these limits, the controller output follows the system surplus.

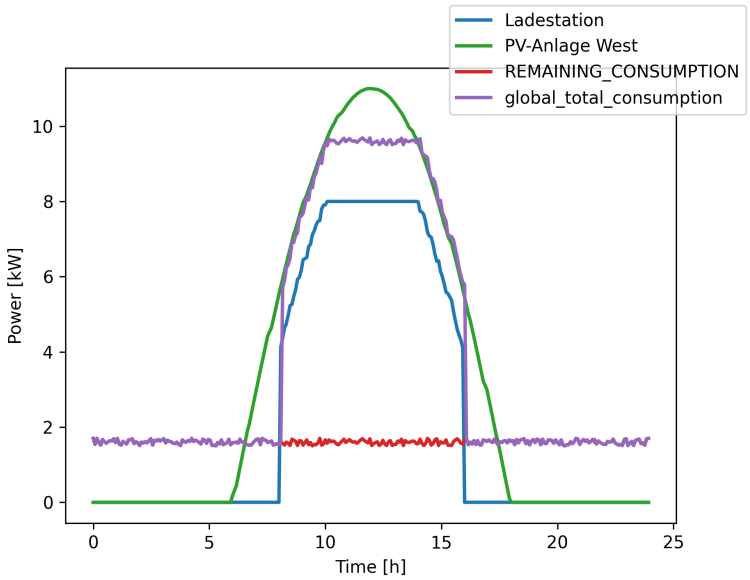

Figure 1.9: EV charging station with *DynamicExcessController* in a simulated OpenCEM system

Coverage Controller

This controller determines the state of the consumer device based on a calculated coverage degree. The coverage degree describes what percentage of the device's power consumption is covered by the surplus. The limit at which the device is turned on can be set. Figure 1.10 demonstrates the functioning of the *CoverageController*. The nominal consumption of the device was set to 3kW in the simulation, and the coverage limit was set to 2, which corresponds to 6kW. It's evident that the simulated electric heater is activated only when there is an excess of more than 7.6 kW (6 kW + 1.6 kW remaining consumption).

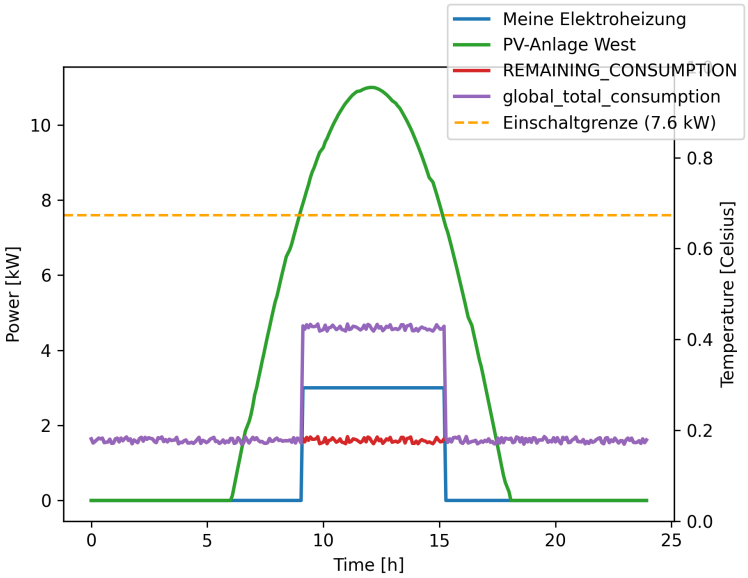

Figure 1.10: Device with *CoverageController* in a simulated OpenCEM system

Price Controller

The price controller determines whether a consumer device should be turned on based on an offered and demanded price. The offer price is calculated based on the consumer's current coverage degree, the grid tariff, and the solar tariff. The demand price is formed from the current actual temperature and a temperature range that can be specified. If the temperature is close to the minimum allowable temperature, the demand price is set higher than the offer price, and the component is turned on. This controller is suitable for heaters and heat pumps.

Figure 1.11 demonstrates the functioning of a *PriceController* using a simulated heat pump. Additionally, there is another device in the system with an *ExcessController*. The dashed blue line in the upper graph depicts the temporal evolution of the simulated temperature, which influences the demand price calculation.

In the lower part of the graph, the temporal evolution of the demand price (consumption price) and the offer price (production price) are shown. The dashed black lines indicate that the heat pump is activated when the demand price is higher than the offer price.

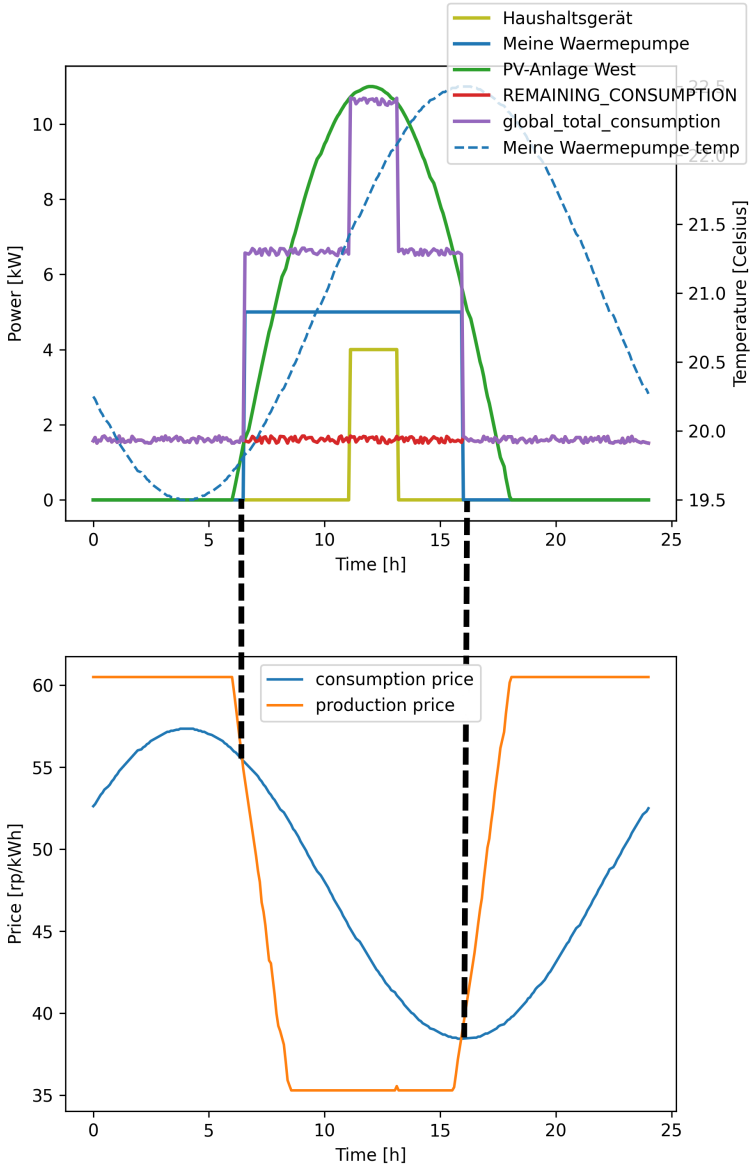

Figure 1.11: Device with an *ExcessController* and a heat pump with a *PriceController* in a simulated OpenCEM system

Between hours 6 and 8, it can be seen that the actual temperature is very close to the set minimum temperature of 19°C. Therefore, the heat pump is turned on even though the consumption is not entirely covered by the surplus.

1.1.5 Devices

A significant part of this semester involved implementing the classes shown in Figure 1.12 within OpenCEM. It's noticeable that all devices inherit from the parent class *Device*. During development, care was taken to implement the concept of polymorphism in object-oriented programming for these devices. This involves providing the same methods to different classes, which may yield different results depending on the device.

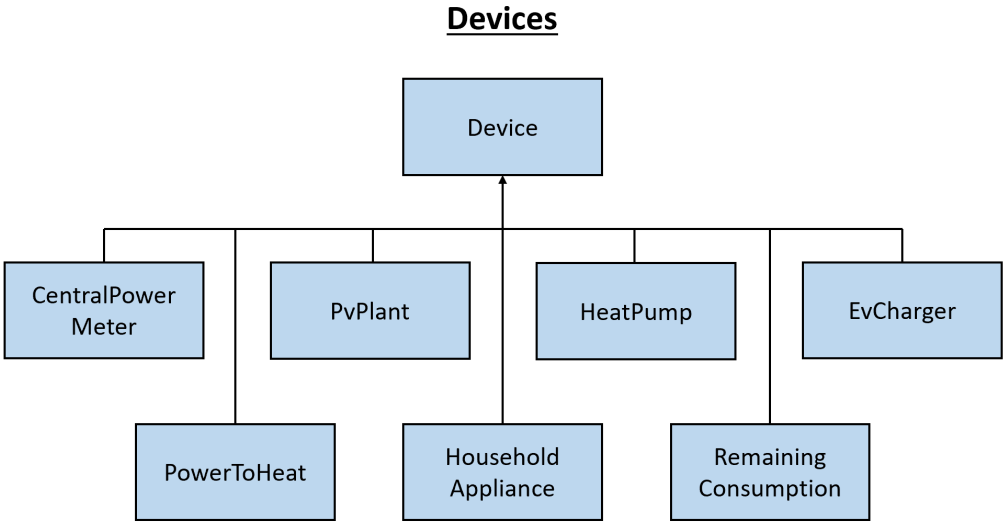

Figure 1.12: Device class structure in OpenCEM

An example of this is the method *calc_controller()*, which calculates the controller of a device and sets it to the desired state. A method is a function that can be applied to instances of a class. All controllable devices have *calc_controller()*, but for example, in the case of an electric vehicle charging station, the vehicle's status needs to be checked, so the method is different from other devices.

Polymorphism allows for creating a flexible and reusable code design. It also helps in making the main program more organized and comprehensible.

The following section briefly describes each device class. Multiple instances of devices can be added to an OpenCEM system, with exceptions for *CentralPowerMeter* and *RemainingConsumption*, where only one instance is allowed.

Central Power Meter

This device represents the connection to the power grid. A bidirectional power meter can be used for this purpose. Alternatively, it's also possible to assign both a total consumption and total production meter to this device.

PV Plant

This device is used for photovoltaic systems. To measure power production, a power measurement device needs to be added to this device. Direct communication with PV panels or inverters hasn't been implemented yet.

Heat Pump

This class is used for heat pumps. Direct communication with heat pumps hasn't been implemented yet, but many heat pumps can be controlled through external terminals, which OpenCEM already supports.

EV Charger

This class is used for electric vehicle charging stations.

For the project, the *wallbe Eco 2.0s* (for development) and *wallbe Pro* (for testing installation) charging stations were available. Both are AC charging stations that can operate on either single-phase or three-phase power. OpenCEM supports both operation modes. They can provide a maximum charging power of 11 kW (Wallbe Eco 2.0s) or 22 kW (Wallbe Pro) with 3-phase operation. The *EV-CC-AC1-M3-CBC-RCM-ETH* charging controller from Phoenix Contact was used in both charging stations, allowing communication through Modbus TCP.

Care was taken while programming this class to make it easily extendable to other charging stations that follow the charging cycle according to *IEC 61851-1*. Figure 1.13 shows the various statuses defined by *IEC 61851-1*.

Fahrzeug-status	Fahrzeug angeschlossen	S2[1]	Ladevorgang möglich	Va[2]	Beschreibung
A	Nein	Offen	Nein	12 V	Vb[3] = 0 V
B	Ja	Offen	Nein	9 V	R2 erkannt B1 (9 V DC): EVSE[4] noch nicht bereit B2 (9 V PWM): EVSE bereit
C	Ja	Geschlossen	Fahrzeug bereit	6 V	R3 = 1,3 kΩ ±3% Belüftung nicht erforderlich
D				3 V	R3 = 270 Ω ±3% Belüftung erforderlich
E	Ja	Offen	Nein	0 V	Vb = 0: EVSE, Kurzschluss am EV Charge Control, Spannungsversorgung nicht verfügbar
F	Ja	Offen	Nein	EVSE nicht verfügbar	EVSE nicht verfügbar

Figure 1.13: Vehicle status according to IEC 61851-1 [2] [**??**]

As the *wallbe Eco 2.0s* and *wallbe Pro* charging stations lack fans, the status D is not present for these models.

Figure 1.14 illustrates the charging process as implemented in OpenCEM.

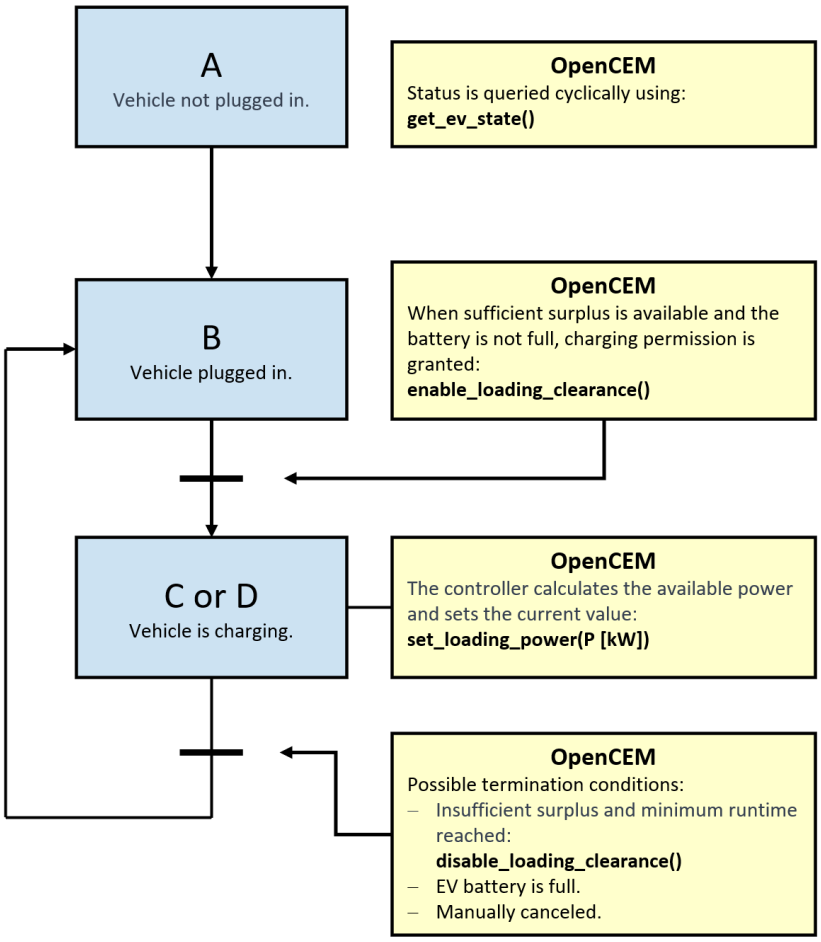

Figure 1.14: Charging process flowchart in OpenCEM

For communication with the charging station, the Modbus registers shown in Table 1.2 are crucial.

Function	Address	Modbus Function	Length	Encoding
System status according to IEC 61851-1	100	Read Input Register	16 Bit	ASCII (A-F)
Charging release	400	Write Coil	1 Bit	1=on, 0=off
Set max charging current	528	Write Holding Register	16 Bit	Integer, e.g., 61 = 6.1A
Read set charging current	300	Read Holding Register	16 bit	Integer, e.g., 61 = 6.1A

Table 1.2: Key Modbus registers of the used charging stations [3]

Since the provided device doesn't have an energy meter, power consumption needs to be calculated based on the set maximum current. The following formula was used:

$$Power\ [\text{kW}] = current\ [\text{A}] * \frac{\text{Number of Phases} * 230V}{1000}$$

Power to Heat

The *PowerToHeat* class can be used for electric heating devices, for example.

Household Appliance

This class can be used for household appliances like washing machines or dryers.

Remaining Consumption

The *RemainingConsumption* device exists virtually. It's used to measure the remaining consumption that cannot be attributed to any other devices in OpenCEM. This can be measured directly using a power meter or calculated.

Protection against Fluctuating Production

Frequent switching on and off significantly impacts the lifespan of electronic devices, especially when there's a short interval between turning them on and off. On cloudy days, the power output of PV panels can fluctuate greatly, leading to repeated switching on and off of components. Therefore, a mechanism was implemented in OpenCEM to counter this effect and protect the devices' lifespan. After a device is turned on, it is locked from being turned off for a specific time. The same applies when a device is turned off and needs to be turned on again.

1.2 supported devices, sensors and actuators

A wide range of devices, sensors, and actuators are already supported in OpenCEM (Figure 1.15). The modular and class-based structure of OpenCEM allows for straightforward addition of new devices.

Sensor / Aktor	Name	Bus-Type	Native	SGr
Power sensor	ABB B23 112 100	Modbus RTU	✅	✅
Power sensor	ABB B23 312 100	Modbus RTU	✅	✅
Power sensor	Shelly 3EM	Shelly local / cloud	✅	-
Relais	Shelly Pro 2PM	Shelly local / cloud	✅	-
Relais	Shelly Pro 4PM	Shelly local / cloud	✅	-
Temp. sensor	Shelly H&T	Shelly cloud	✅	-
Button	Shelly Button 1	Shelly local / cloud	✅	-
Temp. sensor	Shelly TRV	Shelly local / cloud	✅	-
EV charger	wallbe Eco 2.0s and wallbe Pro	Modbus TCP	✅	-

Figure 1.15: Implemented and tested devices, sensors, and actuators in OpenCEM (as of August 18, 2023).

1.3 OpenCEM Main

In this section, we'll delve into the main file of OpenCEM. To start it, you only need to execute the Python script *OpenCEM_mainV2.py*. Instructions on how to do this on a RevPi are provided in Chapter 4.5. The following Figure 1.16 illustrates the functioning of the main file in OpenCEM.

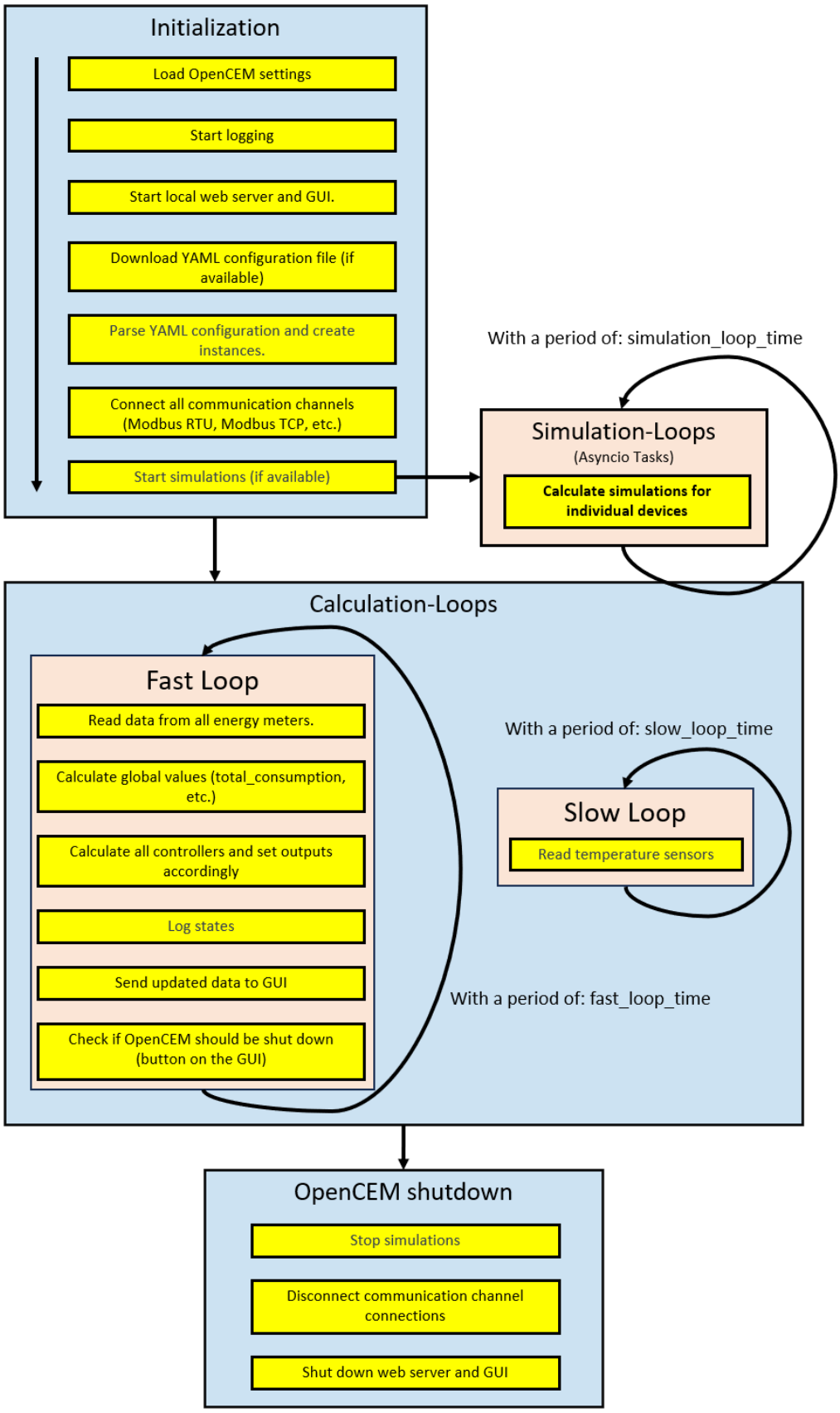

Figure 1.16: Functioning of the *OpenCEM_mainV2.py* script

1.4 OpenCEM GUI

Part of the project assignment was to visualize the key information of devices associated with OpenCEM through a graphical user interface (GUI). The objective was to develop a concept that can dynamically expand, meaning the GUI should be automatically generated for each installation, regardless of the number of devices. Aesthetic design was not the primary focus.

In many IoT products, a GUI is provided via a local web server. This includes products from Shelly as well. By entering the local address of the device, users can access the GUI through a browser to receive information or control the device. Since nearly every device has a browser, this type of GUI is platform-independent and doesn't require additional software or libraries to be downloaded. Furthermore, web technologies allow for creating a dynamic GUI that adapts to the specific installation. Another advantage is that a GUI developed using web technologies can be easily extended in the future.

Due to the mentioned advantages and the existing experience with HTML and JavaScript for website programming, the GUI was created as a local web server. Figure 1.17 depicts the OpenCEM GUI of the test installation in a desktop browser. As it's a local web server, it can be accessed from any device within the local network. The GUI is automatically launched when executing the main file and can be accessed via ***IP-Address-of-the-Device:8000***.

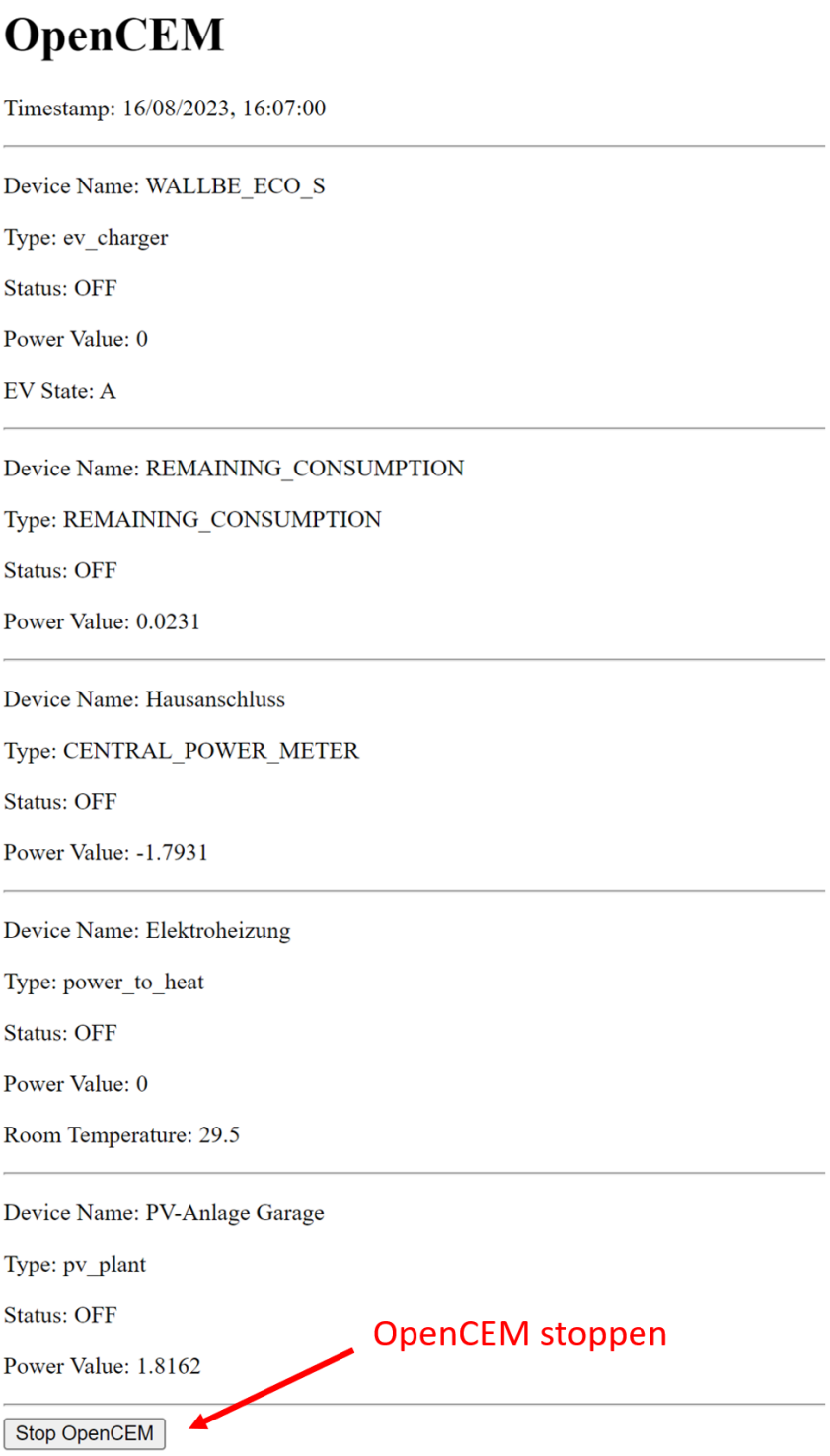

Figure 1.17: OpenCEM GUI of the test installation in a desktop browser

The GUI displays essential information about the devices in the OpenCEM system. If a device has a temperature sensor, the temperature is also shown in the GUI. At the end of the webpage, there's a button labeled *Stop OpenCEM*, which can be used to shut down OpenCEM. After pressing this button, it might take a moment for OpenCEM to shut down, as it checks only once per cycle whether a shutdown is requested.

The web server was created using the *aiohttp Python library*. In Table 1.3, the existing server endpoints and their functions are listed. [4]

Endpoint	HTTP Method	Function
/	GET	Displays the OpenCEM GUI (index.html).
/update	POST	Sends the current OpenCEM data to the server (JSON format).
/latest_data	GET	Returns the latest OpenCEM data as JSON.
/shutdown	POST	Initiates the shutdown of OpenCEM.
/shutdown_requested	GET	Returns whether a shutdown of OpenCEM is requested.

Table 1.3: Existing endpoints of the OpenCEM GUI web server

At the end of each cycle, OpenCEM sends the updated data to the */update* endpoint. The index.html file contains JavaScript that fetches the updated data as JSON from the */latest_data* endpoint every 5 seconds. This data is then unpacked by JavaScript and displayed.

During testing, it was observed that after a crash of OpenCEM, the port of the GUI might remain occupied. To correctly start the web server, the port needs to be released first using the following command:

```
1 npx kill-port 8000
```

1.5 Simulation Mode

As a requirement for OpenCEM, the capability to simulate installations was defined. This allows testing the behavior of OpenCEM before deploying it in a real installation. The simulation mode also served to detect and resolve software bugs early on. When operating with real hardware, there's a risk of causing damage. Conducting simulations beforehand can help mitigate this risk.

For each device from Chapter 1.1.5, a simplified model was created in collaboration with the client. These models aim to mimic the functions of the devices and simulate values such as temperature or current consumption of the device. Each device has a *simulate_device* method. If a device needs to be simulated, this method is automatically executed in a parallel task when creating the device instance. Simulations are halted when OpenCEM is shut down.

With OpenCEM, it's possible to calculate both fully simulated installations and partially simulated configurations. For example, this allows testing the integration of devices (e.g., a second PV plant) into an existing real installation before purchasing and observing the expected behavior. To simulate a device, the *isSimulated* parameter must be set to *true* in the YAML configuration file (See Chap. 2.1). In fully simulated systems, the simulation can be accelerated using the *OpenCEM_speed_up* field (See Chap. 2.2).

To test the behavior of the test installation at the client's location before deployment, a simulation of it was created. This is illustrated in Figure 1.18.

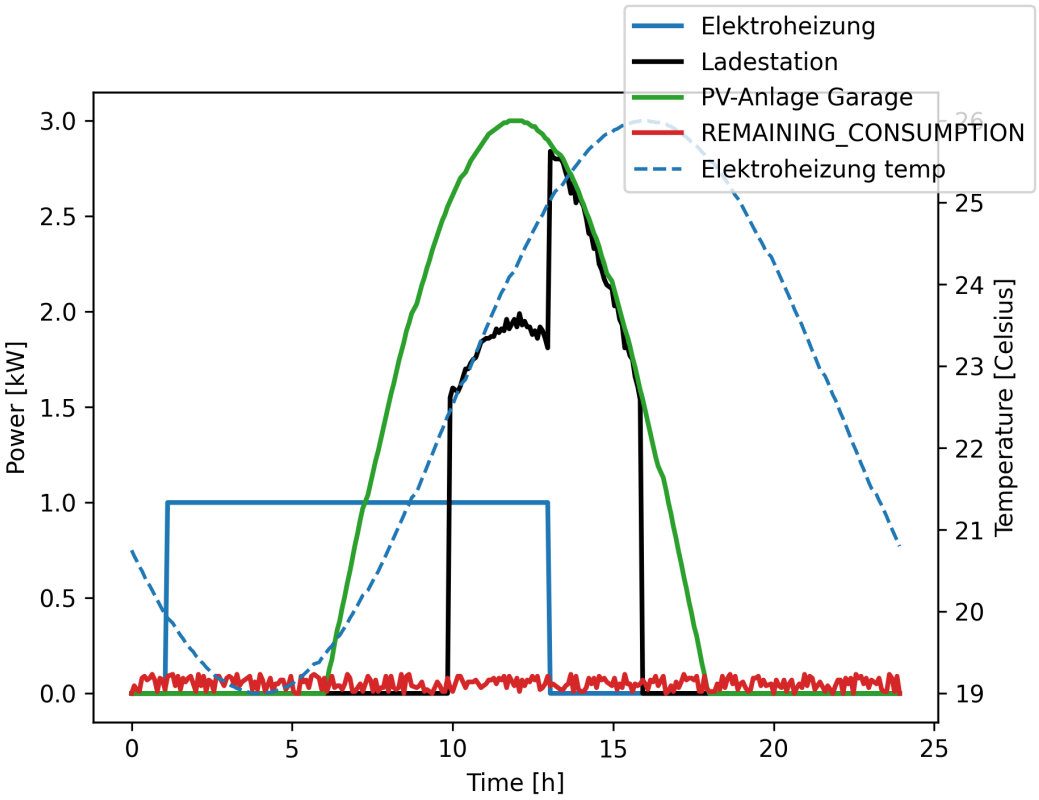

Figure 1.18: Simulation of the test installation

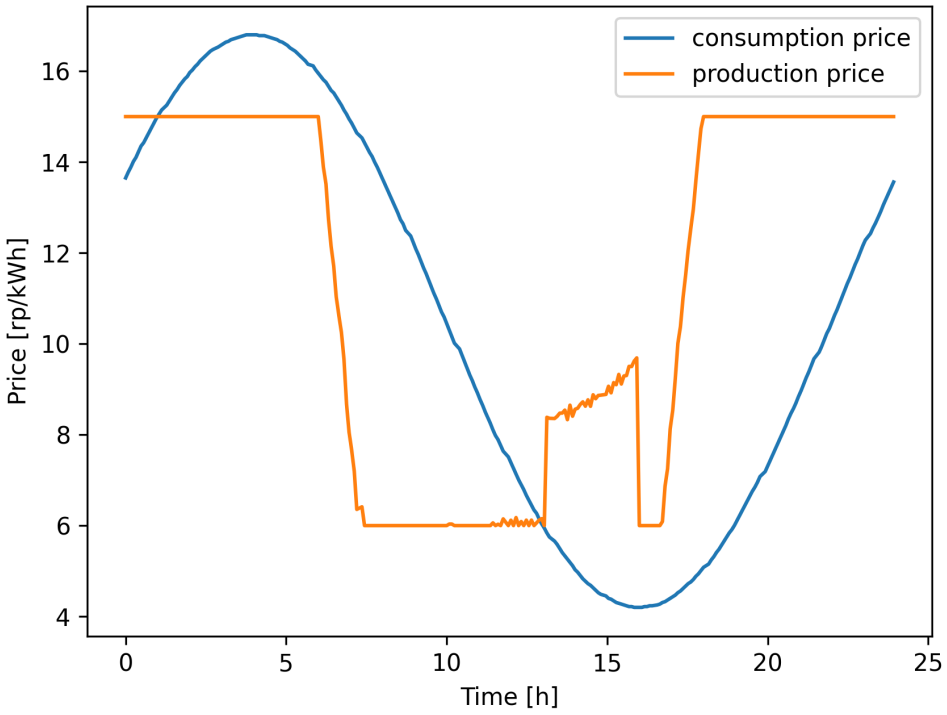

Figure 1.19: Supply and demand price of the price controller from Fig. 1.18

2 OpenCEM Configuration

Since it is planned to make OpenCEM available as open-source software, a high degree of reusability was considered during its development. OpenCEM is designed to serve as a library that can be easily used in various installations. To facilitate this, a schema for YAML configuration files was developed. In this section, the structure of these configuration files will be explained, along with how OpenCEM can be configured and settings can be adjusted. Additionally, in Chapter 2.4, the OpenCEM Configurator (not open-source), developed by the computer science students in the parallel project, will be introduced. This section will also cover the interfaces that connect the Configurator with OpenCEM.

2.1 YAML Configuration File

An OpenCEM system can consist of various devices, communication channels, sensors, actuators, etc., which can be associated with each other. In order to initialize OpenCEM, it's necessary to store this information in a configuration file. YAML is a commonly used data format for this purpose, and it is also used for OpenCEM. YAML uses a minimal syntax, making it easier to read and edit than data formats with complex special characters like XML or JSON. Additionally, data structures are clearly visible in the file, enhancing human readability. This is crucial for OpenCEM since, in the absence of the Web Configurator (Chapter 2.4), configuration files need to be manually created. Another advantage is the support for more complex data structures such as lists. YAML is widely used, so parsing libraries are available for various programming languages. For OpenCEM, the PyYaml library was used.

Basic Structure of the Configuration File

The configuration file should allow the necessary instances (devices, sensors, etc.) to be created during the initialization of OpenCEM. Therefore, it makes sense to define the structure of the YAML file according to the object structures described in Chapter 1. Figure 2.1 illustrates the basic structure of the YAML configuration file. The file is divided into sections: *communicationChannels, actuators, sensors, devices,* and *controllers*. Each of these sections forms a list to which an unlimited number of objects can be appended. This allows the creation of systems of arbitrary size with various combinations. Additionally, the header of the file contains information about the installation, creation time, and version.

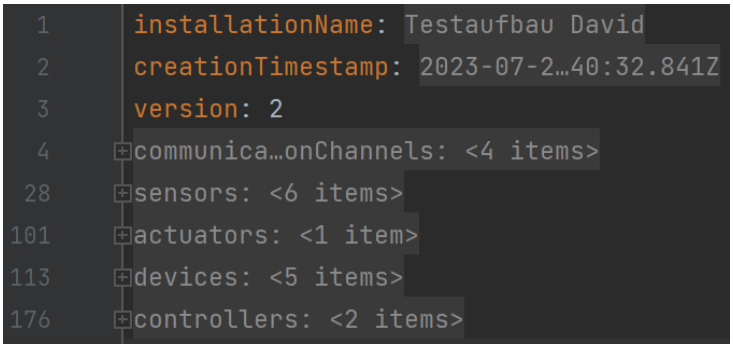

Figure 2.1: Basic structure of the YAML configuration file (Screenshot from PyCharm)

To link objects during parsing, a unique identification of the objects is required. Instances are assigned UUIDs as *id* for this purpose. UUID stands for "Universally Unique Identifier" and is a 128-bit string used to uniquely identify entities. These strings can be generated using UUID generators, which are also available as websites. [5]

The following are examples of configuring communication channels, sensors, actuators, devices, and controllers. These examples aim to illustrate the different concepts to consider during configuration. As OpenCEM supports a wide range of devices, actuators, sensors, etc., these are just a few examples. Additional templates are provided in the folder *YAML_templates* folder.

Figure 2.2 demonstrates how relationships between devices, sensors, etc., are created in the configuration file (highlighted in red). Each instance receives a UUID, through which it can be referenced. This mechanism is exemplified using a sensor and its communication channel.

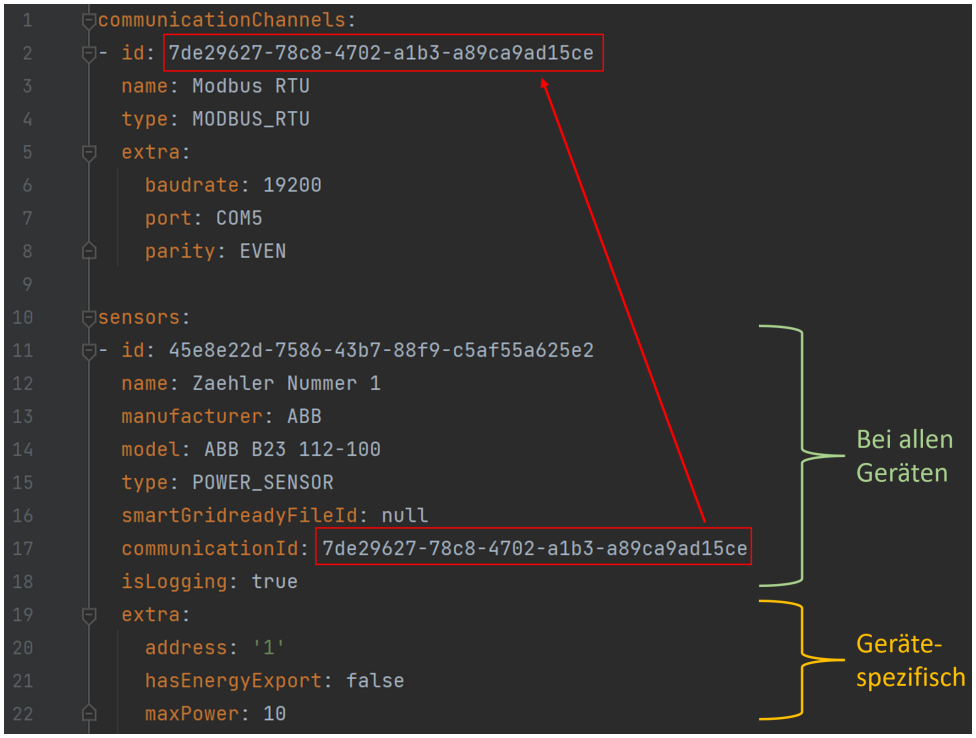

Figure 2.2: UUID referencing and general and device-specific information.

Furthermore, Figure 2.2 depicts the general information shared by each device, sensor, and actuator. Additionally, there are device-specific details that vary between device types. These specific details are captured under *extra*.

In OpenCEM, devices are central, and sensors, actuators, etc., can be associated with them. Figure 2.3 demonstrates the various *id's* (highlighted in red underline) using the example of a heat pump. This pump has different operating modes that can be controlled through combinations of inputs, as shown in the figure. If the *communicationId* field is null, the device cannot communicate directly with OpenCEM and relies on relays. The relay's channels that the device uses are specified under *channels*. The combinations leading to the operating modes, as shown in the figure, are defined under *channelConfig*. The first setting should represent the off state, followed by modes sorted in ascending order of consumption. This can be expanded as needed. Figure 2.4 shows how a *StepwiseExcessController* for this heat pump might look.

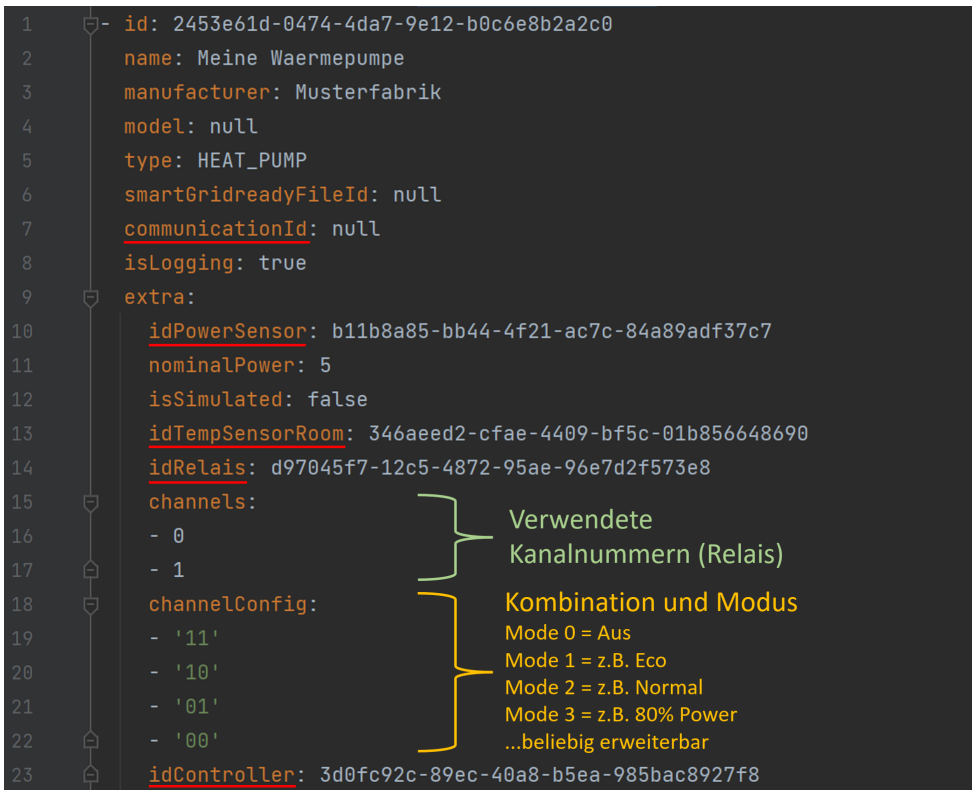

Figure 2.3: Example YAML for a heat pump with multiple operating modes.

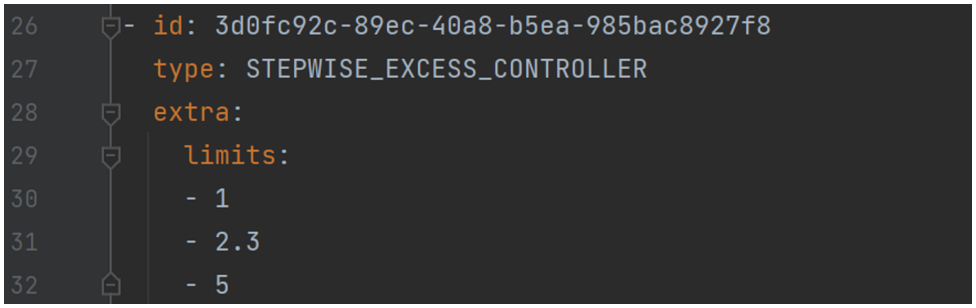

Figure 2.4: Example of a *StepwiseExcessController*.

To simulate devices, the *isSimulated* field can be set to true. Figure 2.5 shows a simulated heat pump with four modes. The simulated consumption for the operating modes can be configured using the *powerDict* field.

```yaml
- id: 2453e61d-0474-4da7-9e12-b0c6e8b2a2c0
  name: Waermepumpe
  manufacturer: Musterfabrik
  model: null
  type: HEAT_PUMP
  smartGridreadyFileId: null
  communicationId: null
  isLogging: true
  extra:
    nominalPower: 5
    isSimulated: true
    idController: 3d0fc92c-89ec-40a8-b5ea-985bac8927f8
    powerDict: # add if simulated and multiple modes. Describes consumption in this mode
      0: 0
      1: 1.5
      2: 3.5
      3: 5.5
```

Simulierter Verbrauch für den jeweiligen Modus (in kW)

Figure 2.5: YAML for a simulated heat pump.

2.2 YAML OpenCEM Settings

To adjust the functionality of the OpenCEM Main Loop (1.3), another YAML file is used. This file is located in the *yaml* folder and is named *OpenCEM_settings.yaml*. The filename and location must not be changed for proper functioning. Figure 2.6 provides an example of possible OpenCEM settings. The entries in this file will be explained in more detail in the following section.

```yaml
OpenCEM_speed_up: 1 # put 1 here if the system is not simulated

# time for the different loops in seconds
fast_loop_time: 20
slow_loop_time: 60
simulation_loop_time: 5

# the time the OpenCEM should run for in seconds
duration: 0 # 0 means will run forever

# settings for logging
log_events: true
log_stats: true
console_logging_level: 20

# credentials for loading configuration
installation: "installation name"
password: "password"
token: "token"
backend_url: "https://cem-cloud-p5.ch/api"

# keep the path empty if you download the yaml from the server
path_OpenCEM_config: "yaml/Testaufbau_David_V2.yaml"
```

Figure 2.6: Example of OpenCEM_settings.yaml

OpenCEM_speed_up

If you want to test a system with solely simulated devices, you can set the *OpenCEM_speed_up* parameter to determine how many times faster OpenCEM should run compared to reality. If OpenCEM is used in a real installation or an installation with partially simulated devices, this parameter should be kept at 1. Setting this parameter too high may lead to calculations not being completed in the desired time or errors occurring. The maximum possible value depends on the chosen cycle times (explained in the next section) and the available computing power. If errors occur, cycle times can be increased or the *OpenCEM_speed_up* parameter can be reduced.

Cycle Times and Duration

As explained in Chapter 1.3, OpenCEM has different loops with different cycle times. The parameters *fast_loop_time*, *slow_loop_time*, and *simulation_loop_time* can be used to adjust these cycle times. The cycle times are specified in seconds.

To conduct tests and create simulations, it's useful to be able to set the duration for which OpenCEM should run. The duration is specified using the *duration* parameter (in seconds). To run the system endlessly, this parameter should be set to 0.

Logging

The *log_events* and *log_stats* parameters can be used to enable or disable the loggers mentioned in Chapter 3. When testing the system, it was advantageous to display generated log messages directly in the console. The *console_logging_level* parameter can be used to set the logging level that should be displayed in the console.

Server Credentials and YAML Path

If a device configuration has been created for an installation in the Web Configurator, the access credentials can be specified here. Section 2.4 provides detailed information about the interface between OpenCEM and the Web Configurator.

If you are exclusively using the open-source functionalities of OpenCEM, you can provide the local path to the YAML configuration file using the *path_OpenCEM_config* parameter.

2.3 SmartGridready Devices

Devices that support SmartGridready can be read in through their XML description files. In the YAML configuration file, the path to the XML can be specified, as shown in Figure 2.7. Alternatively, the OpenCEM Web Configurator can be used to add SmartGridready description files (not open-source). As illustrated in Figure 2.8, a UUID can be inserted into the device's *smartGridReadyFileId* field. Upon starting OpenCEM, the required XML files will be automatically downloaded.

SmartGridready files for the ABB B23 112-100 and ABB B23 113-100 energy meters were tested.

```
- id: aa2c4f9a-54e8-4959-acca-9ad626f7831e
  name: Meter Consumption Total
  manufacturer: ABB
  model: ABB B23 112-100
  type: POWER_SENSOR
  smartGridreadyFileId: "xml_files/SGr_04_0016_xxxx_ABBMeterV0.2.1.xml"
  ...
```

Figure 2.7: Example of an SGr file specified using a local path.

```
- id: aa2c4f9a-54e8-4959-acca-9ad626f7831e
  name: Meter Consumption Total
  manufacturer: ABB
  model: ABB B23 112-100
  type: POWER_SENSOR
  smartGridreadyFileId: de7b4c9d-d7ef-4d87-b676-fe51e60ecf4b
  ...
```

Figure 2.8: Example of an SGr file specified using a UUID.

2.4 OpenCEM Web Configurator

The CEM cloud is a possible extension to the local running code, which in future can be used to automatically generate the configuration YAML files using a web-based configurator. Figures 2.9 and 2.10 show some excerpts from the developed web platform.

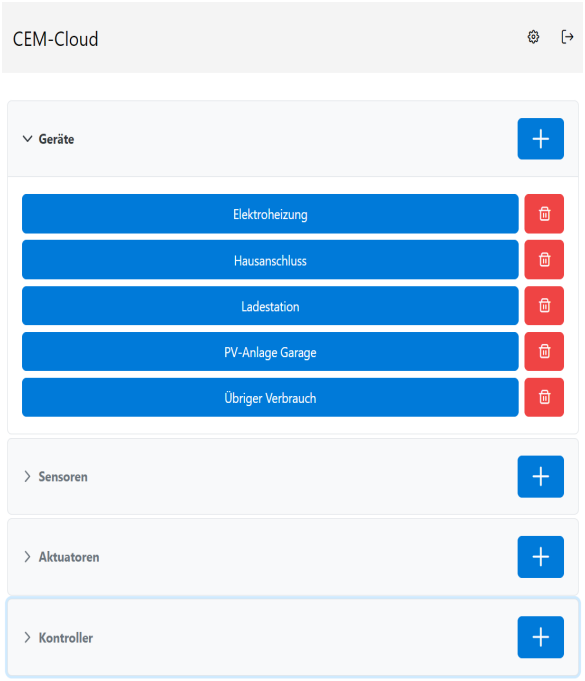

Figure 2.9: Overview of an installation in the Web Configurator

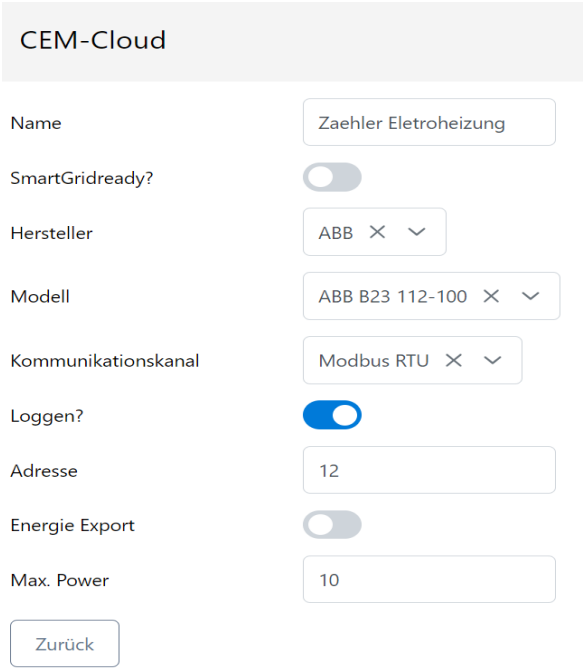

Figure 2.10: Configuring a Power Meter

Figure 2.11 illustrates the interface between the CEM Cloud and OpenCEM.

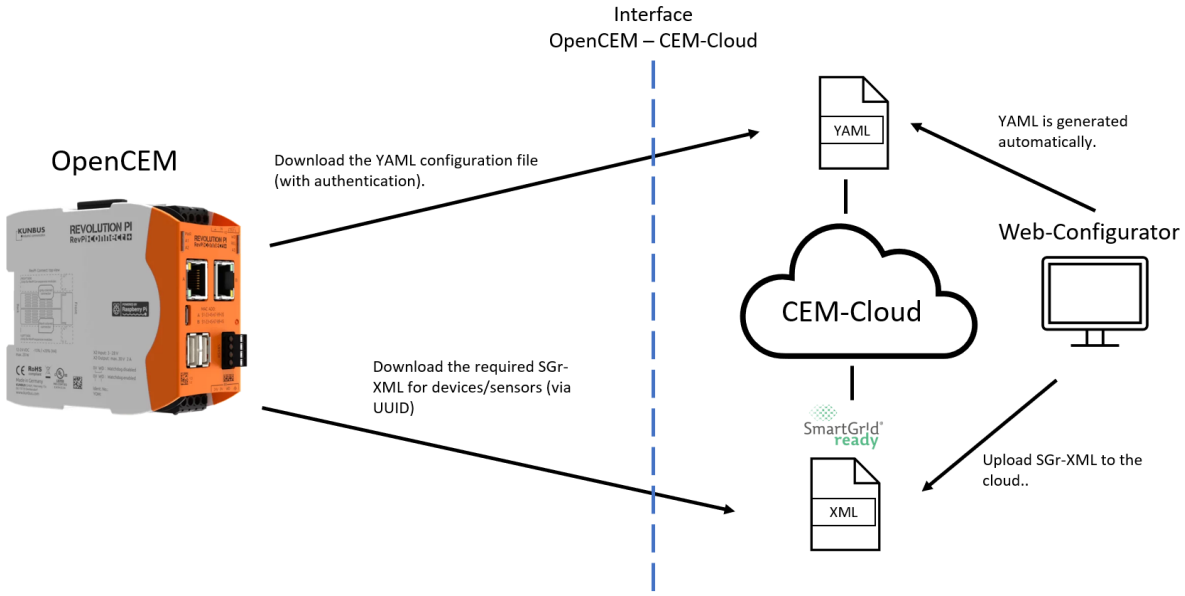

Figure 2.11: Interfaces between CEM Cloud and OpenCEM

The main interface between the two projects is the YAML configuration file (Chapter 2.1), which can be automatically generated using the web tool and subsequently loaded into OpenCEM. Configurations can also be created without using the web configurator, but this requires a certain level of expertise and is less user-friendly. he CEM-Cloud is described in a separate document. [6]

Furthermore, the web configurator allows assigning a Smart Grid-Ready file to a device. An administrator can upload XML files through the website, which are then available for selection to CEM Cloud users. Figure 2.12 demonstrates the configuration of a power meter using an SGr XML file.

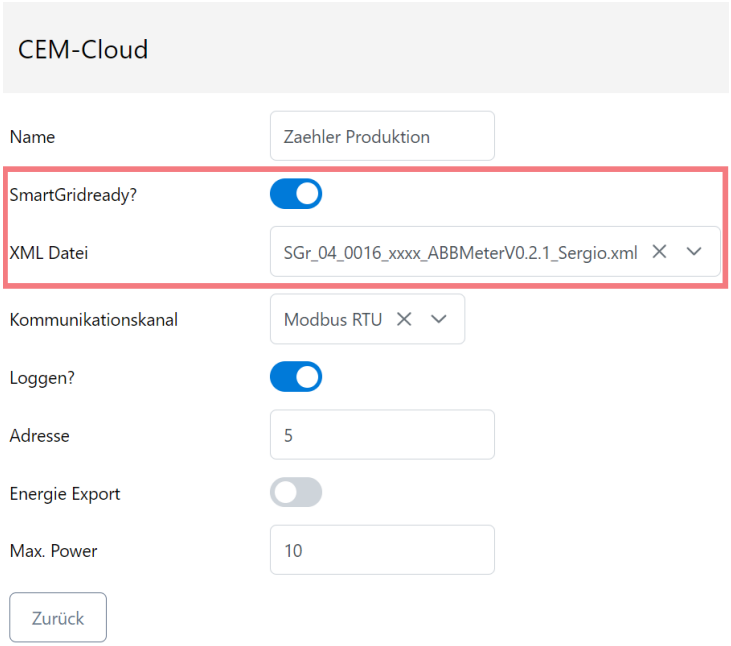

Figure 2.12: Configuration of a Power Meter using an SGr File

3 Logging

Logging refers to the process of recording information or events during the runtime of a program. It is used to better understand the behavior of the software and can be a valuable tool for identifying the causes of errors. Logs can also be used to gather statistics about the system, which can be visualized for better comprehension if necessary.

For OpenCEM, the requirement was to develop logging that captures essential information about devices and records system events (e.g., warnings). Much of the logging was already implemented in the previous semester, with a focus on system event logging and logging for sensors and actuators. These loggers remained unchanged for P6 and were covered in the P5 project report.[1]

In the following section 3.1, the newly added logging for devices will be discussed.

3.1 Device Statistics Logger

An essential part of P6 was the implementation of the device classes from Chapter 1.1.5. Since devices are at the core of OpenCEM and are linked to sensors, actuators, and controllers, they provide an overview of the system's crucial data. These data points need to be recorded, which is why a new logger for devices was developed.

This logger is named *statistics_logger_devices()* and can be initiated using the command *create_statistics_logger_devices()*. To add device states to the log, the method *log_values()* can be executed. In the OpenCEM main file, this is done for all devices present in the system at the end of each iteration. Since this logger also contains information about sensors and actuators, the statistics logger from the previous semester is only needed when recording sensors or actuators without associated devices.

Additionally, this log includes extra global information. Examples of this information are the overall total production or consumption. If a price controller is used for a device, the supply and demand prices are also added to the log.

To facilitate visualization of the log (e.g., using Excel), similar to the loggers from P5, the log is stored in CSV format here as well. Entries are directly saved upon creation and saved as dated files in the *_logFiles* folder at midnight. This direct saving ensures that even in case of a power outage, the most recent log entries are preserved.

3.2 Creating Graphs

To better analyze device statistics, it's important to visualize them. For this purpose, a Python script was developed (*plot_stats.py*). This script takes a local path to a log file as input, along with entries that should not be included in the graph. Additionally, various settings can be adjusted, such as the color of individual plots, to generate the desired graph. The Python script's comments provide details about the different configuration options.

This script can be used to visualize both simulated and real OpenCEM systems. All graphs in this document were generated using the mentioned Python script.

4 Commissioning

In this chapter, we will discuss the commissioning of the test installation at the client's site. The RevPi Connect+ by Kunbus was used for the real installation. The following sections are intended to serve as an instruction manual for successfully installing and starting OpenCEM on a RevPi device.

4.1 Factory Reset

To avoid complications, the RevPi Connect+ was reset to a fresh version of the operating system, also known as an image. These images are provided and updated by Revolution Pi on their website. However, not all versions are supported on all devices. The supported versions for different RevPi devices are shown in Figure 4.1.

Device	Image
RevPi Core 1	Wheezy, Jessie, Stretch, Buster
RevPi Core 3	Jessie, Stretch, Buster, Bullseye
RevPi Core S / RevPi Core SE	Buster, Bullseye
RevPi Core 3+	Wheezy, Jessie, Stretch, Buster, Bullseye
RevPi Connect	Jessie, Stretch, Buster, Bullseye
RevPi Connect S / RevPi Connect SE	Buster, Bullseye
RevPi Connect+	Jessie, Stretch, Buster, Bullseye
RevPi Connect 4	Bullseye
RevPi Compact	Stretch, Buster, Bullseye
RevPi Flat	Buster, Bullseye

Figure 4.1: Supported images for RevPi devices [7]

For the purpose of setting up the RevPi Connect+ from scratch, the latest supported version (2023-06-26-revpi-bullseye-arm64) was installed. The installation procedure was followed using the instructions provided on the Revolution Pi website. [8]

4.2 Verbindung und VNC

To communicate with the RevPi Connect+, you first need to identify its IP address. To do this, connect the device to a 24V DC power supply and connect it to your home network using an Ethernet cable. Then, you can use an IP scanner to find the IP address of the device. For this project, the freely available *Advanced IP Scanner* was used. In Figure 4.2, the IP addresses of the devices relevant to this project are displayed.[9]

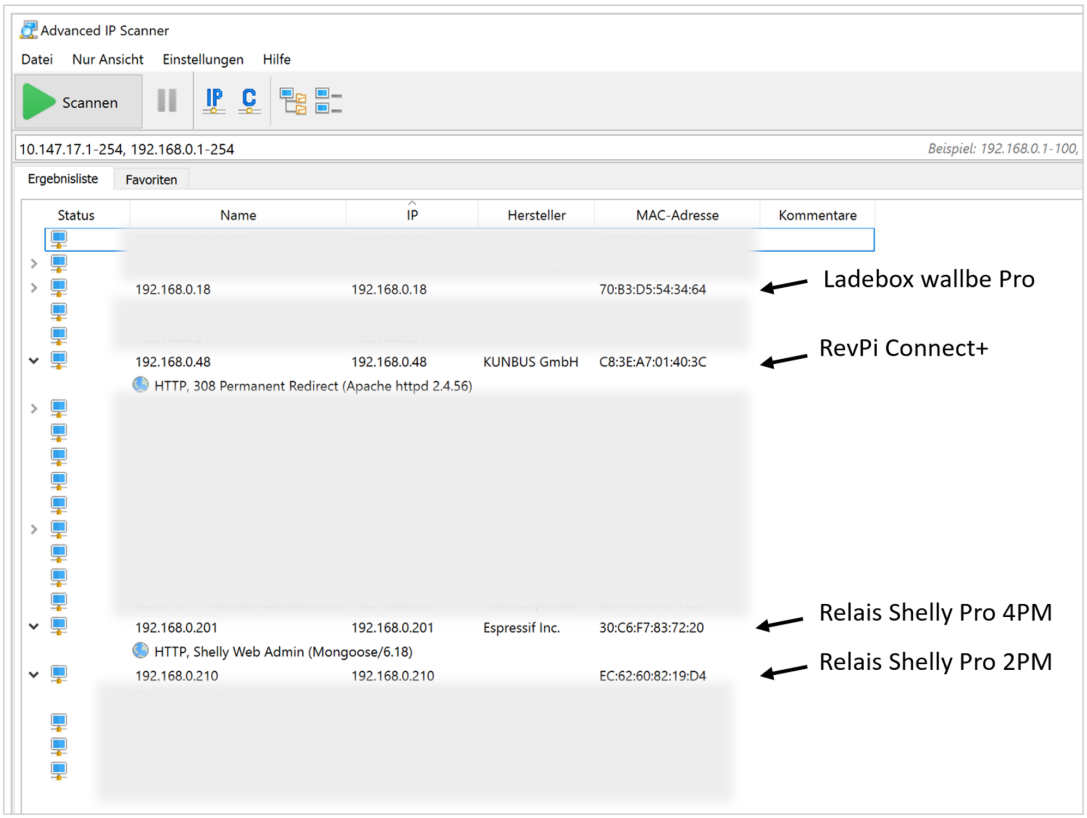

Figure 4.2: Network IP scan

Once the IP address of the RevPi Connect+ is known, you can establish a connection to the device console using an SSH terminal. The free software *PuTTY* was used for this purpose. [10] As shown in Figure 4.3, you can create a connection to the device using its IP address. Then, the user will be prompted for the username and password, which can be found on the RevPi device's page.

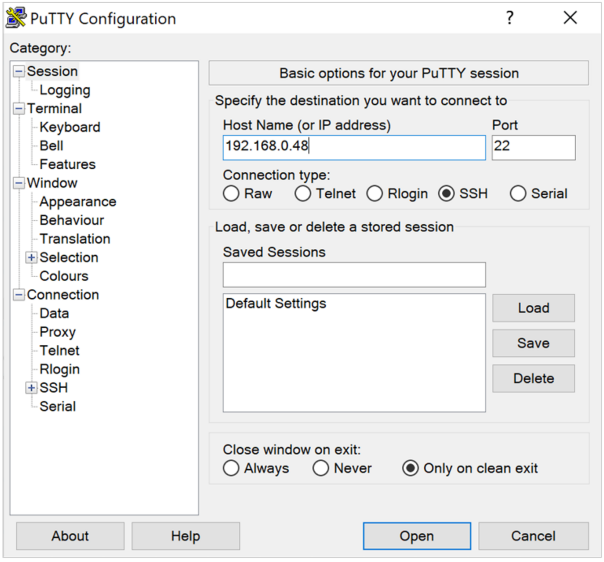

Figure 4.3: Connecting to RevPi Connect+ using PuTTY.

Next, the graphical user interface is activated for the RevPi device. This step is taken because console-based operation is not very user-friendly, and the OpenCEM should be installable and usable by less experienced users. As some RevPi devices lack video outputs (e.g., RevPi Flat) and remote access for OpenCEM is desired, it's suitable to install VNC software on the RevPi Connect+. VNC stands for *Virtual Network Computing*. It's a technology that allows you to control the screen content and inputs of a computer either locally or over the internet. Raspberry Pi recommends using the RealVNC software. [11] Enabling the graphical interface and VNC software can be done in a combined step, as shown in Figure 4.4 to Figure 4.7.

```
sudo apt update
sudo apt install realvnc-vnc-server
```

Figure 4.4: Installing RealVNC on the RevPi device.

```
sudo raspi-config
```

Figure 4.5: Open Raspberry Pi Config

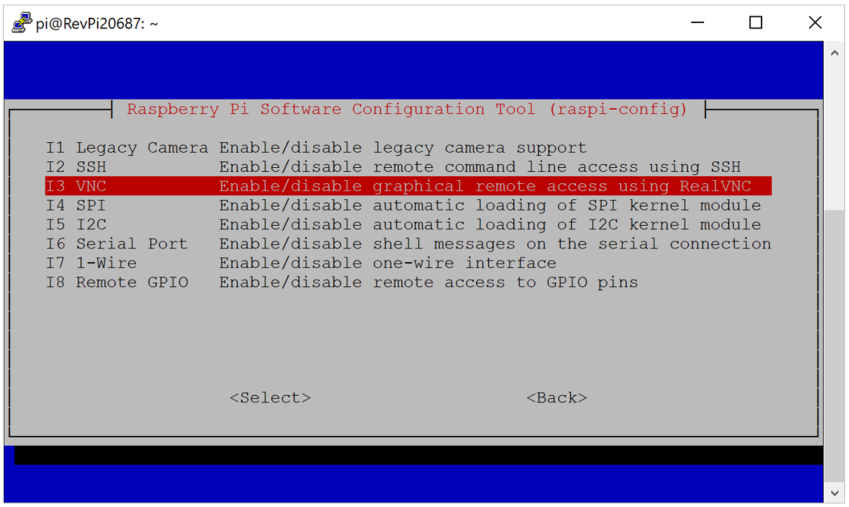

Figure 4.6: Activating VNC in Raspberry Pi Config

```
sudo reeboot
```

Figure 4.7: Raspberry Pi reboot command

Remote Access

RealVNC also provides the capability to access devices over the internet, where the RealVNC Server software is installed. This was necessary for this project to access the OpenCEM of the test installation remotely from home. For this purpose, an account was created with RealVNC. The free version allows adding up to 3 devices to an account. Figure 4.8 illustrates how remote access can be enabled on the RevPi. First, the VNC settings menu should be accessed from the RevPi's taskbar (marked with a red arrow). Then, with the newly created account, you can log

in and proceed with the remote access installation. Afterward, you can view the device's screen over the internet using the VNC Viewer.

Figure 4.8: Enabling remote access using VNC Server on the RevPi Connect+.

4.3 Python Update

As explained in Section 4.1, the process of loading the new operating system onto the RevPi device was discussed. This operating system comes with the pre-installed Python version 3.9.2. However, since the OpenCEM uses the SmartGridready library developed and tested on Python version 3.10.2, an update of the Python version on the RevPi Connect+ was necessary. To achieve this, the provided installation guide was followed. [12]

Python version 3.10.2 was successfully installed. All existing versions of Python were retained on the RevPi Connect+ as they might be used by other software applications on the device.

4.4 GitHub on RevPi

As OpenCEM is intended to be available as open-source software in the future, GitHub offers a straightforward way to make the software available for download. The operational details of GitHub and how it was utilized in this project are explained in more detail in Chapter **??**.

To be able to use GitHub on the RevPi device, Git needs to be installed first and then linked to your GitHub account. The following instructions demonstrate how this can be achieved.

Git auf RevPi-Gerät installieren. [13]

```
sudo apt-get update
sudo apt-get install git-all
```

GitHub auf RevPi-Gerät installieren. [14]

```
type -p curl >/dev/null || (sudo apt update && sudo apt install curl -y)
curl -fsSL https://cli.github.com/packages/githubcli-archive-keyring.gpg | sudo dd
    of=/usr/share/keyrings/githubcli-archive-keyring.gpg \
&& sudo chmod go+r /usr/share/keyrings/githubcli-archive-keyring.gpg \
&& echo "deb [arch=$(dpkg --print-architecture)
    signed-by=/usr/share/keyrings/githubcli-archive-keyring.gpg]
    https://cli.github.com/packages stable main" | sudo tee
    /etc/apt/sources.list.d/github-cli.list > /dev/null \
&& sudo apt update \
&& sudo apt install gh -y
```

To link Git with your GitHub account, enter the following command. Subsequently, a webpage will open where you can log in to authenticate the device. [15]

```
gh auth login
```

The software for OpenCEM is currently stored in a private GitHub repository, which is why authentication with GitHub is required to access it.

4.5 OpenCEM Installation

To download the OpenCEM from the GitHub repository, you can use the following command, where *url-to-repository* is the internet address of the GitHub repository.

```
git clone "url-to-repository"
```

The relevant branch for installation is called *OpenCEM_Main*. In it, all the necessary files required for a functioning OpenCEM are included. The other branches were used for the development and testing of OpenCEM.

Navigate to the OpenCEM repository folder.

```
cd OpenCEM/SGrPython/
```

Switch to the OpenCEM_Main branch.

```
git checkout OpenCEM_Main
```

To ensure the correct functionality of OpenCEM, several external Python libraries are required. For this purpose, you can create a virtual Python environment (virtual environment) in which the necessary libraries can be installed. A virtual Python environment is an isolated environment from other projects. This ensures that the correct versions of libraries are used and a clean environment for OpenCEM is created, making maintenance easier later. To do this, navigate to the OpenCEM folder in the system console and enter the following commands.

Create and activate the virtual Python environment.

```
python3 -m venv myenv
source myenv/bin/activate
```

The required external libraries are listed in the file *requirements.txt* and can be downloaded and installed using the following console command.

```
pip install -r requirements.txt
```

Starting OpenCEM

Once all the external Python libraries are installed, and the YAML configuration file has been created for the desired installation, you can start OpenCEM using the following commands.

Navigate to the OpenCEM folder and activate the virtual Python environment.

```
source myenv/bin/activate
```

Start OpenCEM.

```
python3.10 OpenCEM_mainV2.py
```

Before testing OpenCEM in the real installation, a simulation was run on the RevPi Connect+. This simulation worked correctly, allowing the continuation of the setup of the real system. [**??**]

4.6 OpenCEM Autostart

To ensure that OpenCEM automatically starts again after a power outage, an autostart for OpenCEM on the RevPi has been configured. The following instructions explain this process in detail.

First, a folder named *autostart* needs to be created in the hidden folder */home/pi/.config*. In this newly created folder, the file *autostart.desktop* should be created. The content of the file should look as follows:

autostart.desktop file:

```
[Desktop Entry]
Exec=sudo bash /home/pi/OpenCEM/on_reboot.sh
```

Next, in the folder */home/pi/OpenCEM/*, create the new file *on_reboot.sh* with the following content. on_reboot.sh file:

```
#!/bin/bash
lxterminal --title="OpenCEM" -e "cd OpenCEM/SGrPython; source myenv/bin/activate; python3.10
    OpenCEM_mainV2.py"
```

Now, the RevPi can be rebooted. OpenCEM will automatically start during the boot-up process.

4.7 Test Setup

This chapter describes the test setup that was installed in the client's garage. Figure 4.9 shows the schematic of the real test installation.

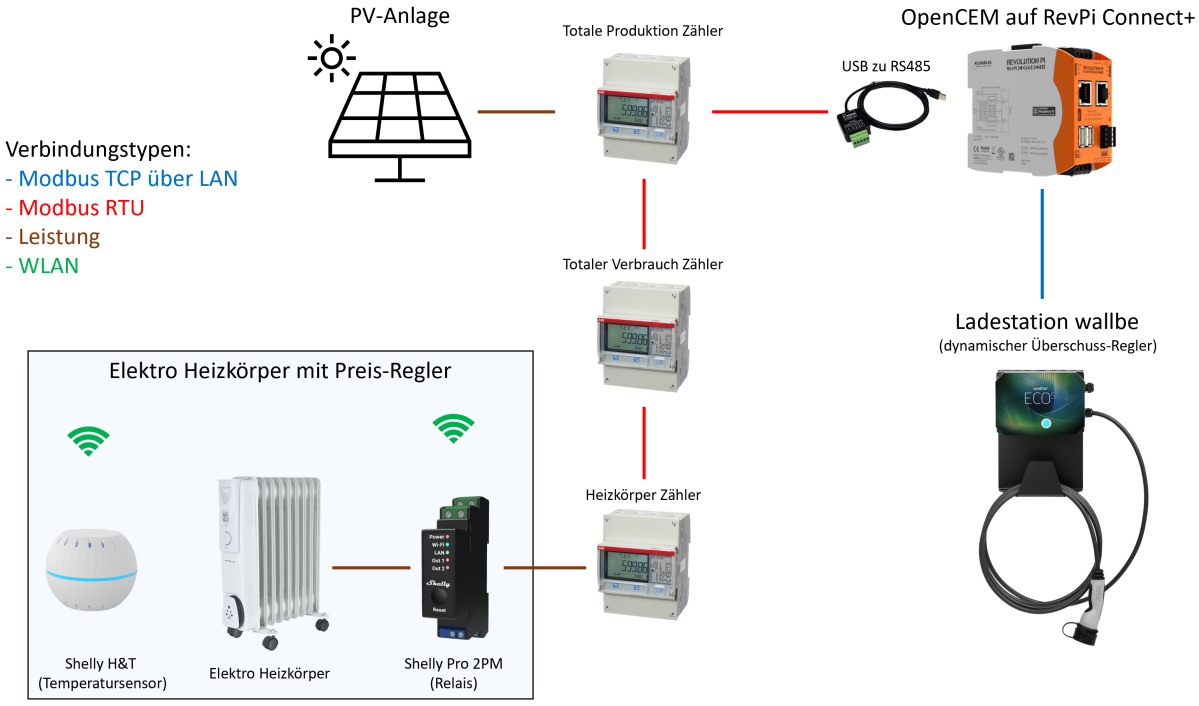

Figure 4.9: Schematic of the test installation

For the commissioning, the test setup was prepared so that it only needed to be powered via the input terminals and the electric heater had to be connected via the output terminals. The ABB energy meters for total consumption and production were already installed at the client's premises and needed to be connected to the RTU network. The RevPi Connect was integrated into the RTU network as the master via a USB to RS485 adapter.

Subsequently, the RevPi was connected to the existing switch via an Ethernet cable and thus integrated into the LAN. The wallbe charging station was also connected to this switch, establishing a connection between the RevPi and the charging station.

Some devices in the test installation required a Wi-Fi connection. As there was poor Wi-Fi coverage in the garage, a temporary Wi-Fi router was installed. The Shelly devices were then added to the Wi-Fi network using the Shelly app. This process was already covered in the P5 project report.[1]

In Figure 4.10, you can see the real installation at the client's site. The components outlined in green were used for testing OpenCEM, while the ones marked in red were not used. The test setup (Fig. 4.11) was placed on a table at the installation site and connected to the installation.

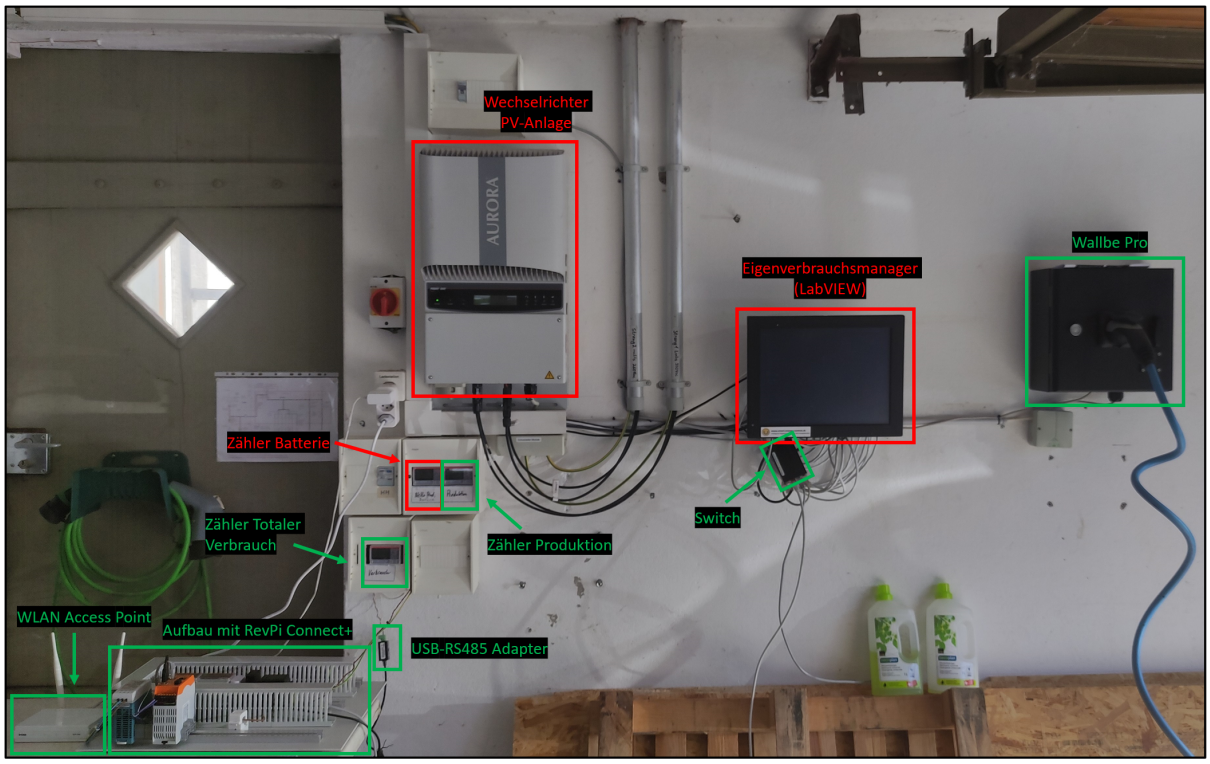

Figure 4.10: Real installation with marked used devices.

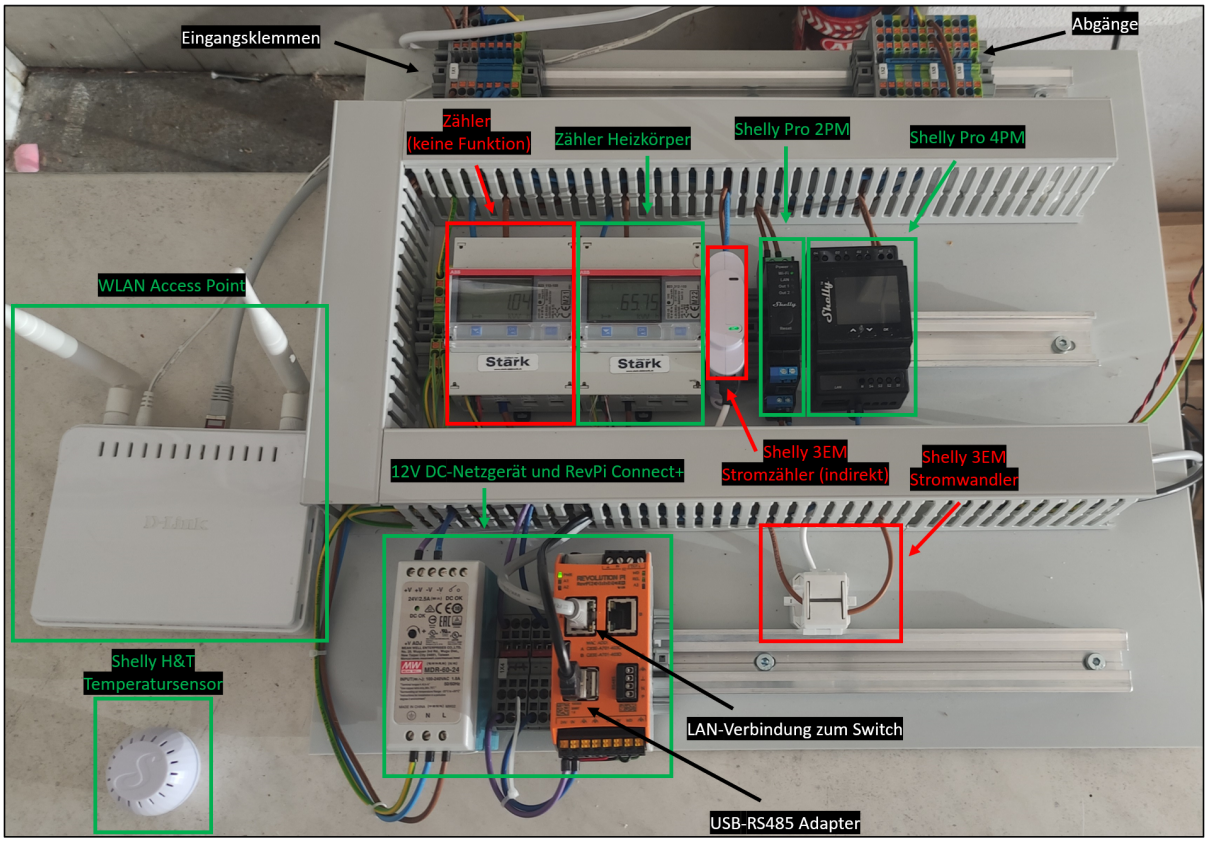

Figure 4.11: Test setup with marked used devices.

Once the hardware was connected, the network addresses of the devices were identified. The Shelly devices' network addresses are found in the Shelly app. For the charging station, an IP scan of the network was performed, and the Modbus RTU energy meters' addresses can be read or changed on the hardware. Table 4.1 shows the network addresses of the devices. There were additional devices in the networks that were not used for the test installation.

Network	Device	Address	Function
Modbus RTU	ABB B23 112-100	5	Production meter
Modbus RTU	ABB B23 112-100	1	Total consumption meter
Modbus RTU	ABB B23 112-100	12	Electric heater meter
Modbus RTU	ABB B23 312-100	3	Battery meter (not used)
Modbus RTU	ABB B23 112-100	11	No function
LAN	RevPi Connect+	192.168.0.48	Executes OpenCEM
LAN (Modbus TCP)	wallbe Pro	192.168.0.18	Charging station
WLAN	Shelly Pro 2PM	192.168.0.210	Can control electric heater
WLAN	Shelly Pro 4PM	192.168.0.201	Can control RevPi
WLAN	Shelly 3EM	192.168.0.26	RevPi meter (not used)
WLAN (Shelly Cloud)	Shelly H&T	701f93	Room temperature sensor

Table 4.1: Network addresses of the devices

The RevPi Connect+ could be controlled via the Shelly Pro 4PM relay. This feature was integrated into the test setup to provide a restart option by toggling the relay via the Shelly Cloud in case the RevPi didn't respond to remote access requests.

To test the communication with the devices, the Python script *communication_testing.py* was executed on the RevPi. This script takes a YAML configuration file of an installation as input. It then checks and evaluates the connections to the devices. The connection test was successfully performed at the test installation.

5 Error Handling

Another sub-goal of this project was to develop a concept for the error handling of OpenCEM and to implement it. It is important that errors are recorded, warnings are displayed to the user, and the crashing of OpenCEM is prevented.

To achieve this, code was implemented to handle potential errors. Table 5.1 shows the error cases for which software was implemented in OpenCEM.

Error Case	**Tested**	**Result**
Disconnect USB-RS485 adapter from RevPi	X	The adapter works again after reconnecting, and values can be read from the devices (Fig. 5.1).
Turn off Wi-Fi	X	Devices that require Wi-Fi automatically reconnect after it's turned on again (Fig. 5.2).
Incorrect CEM Cloud login information entered	X	OpenCEM is stopped, and the user is prompted to verify login information (Fig. 5.3).
Power outage	X	OpenCEM is automatically restarted when the RevPi restarts.

Table 5.1: Potential error cases to be handled.

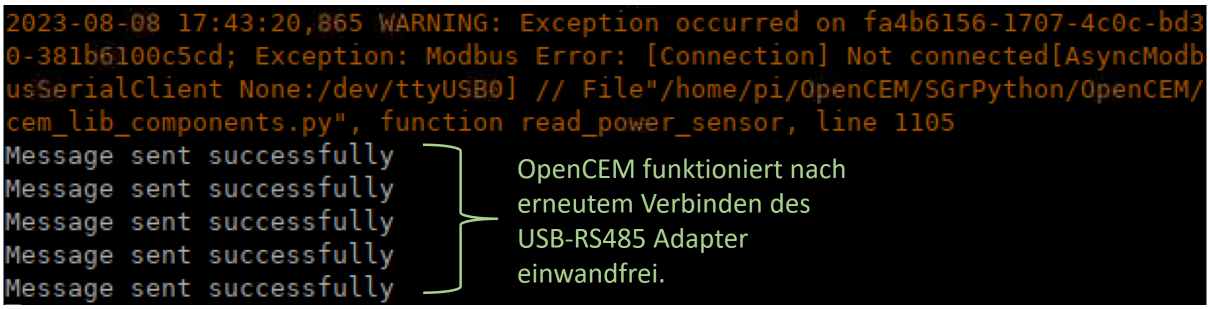

Figure 5.1: USB-RS485 adapter disconnected from RevPi.

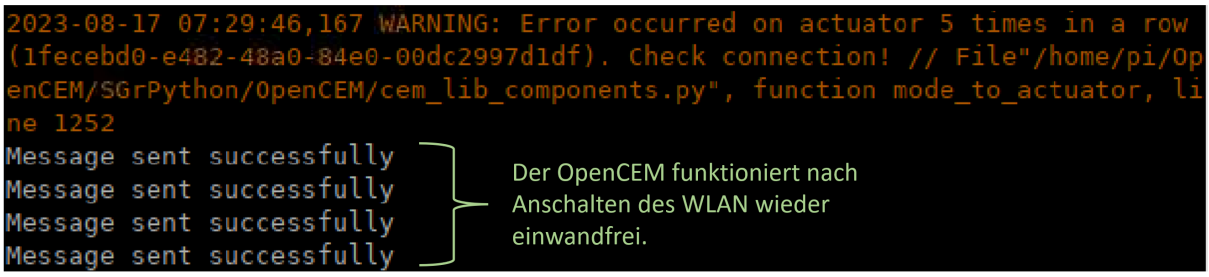

Figure 5.2: Wi-Fi temporarily turned off.

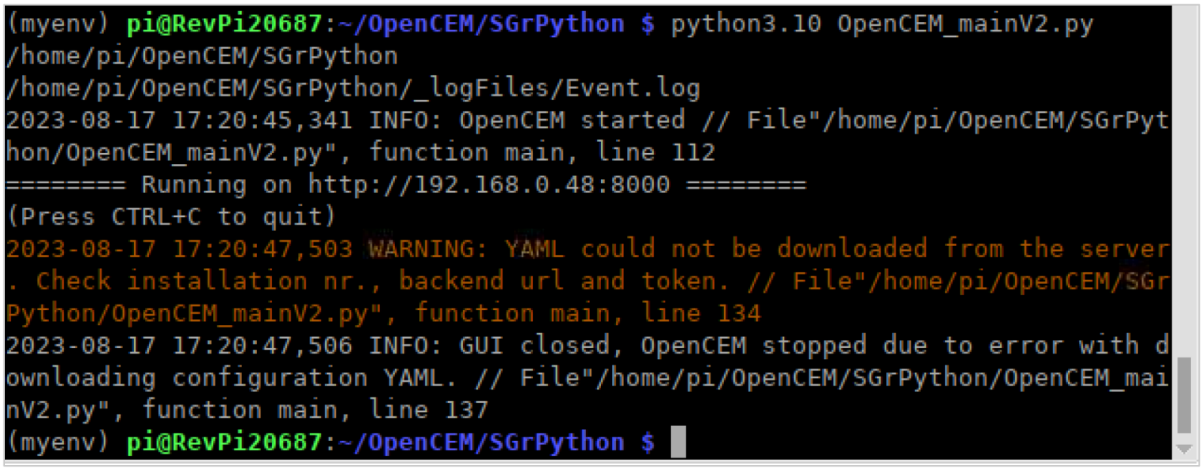

Figure 5.3: Incorrect login credentials entered for CEM Cloud.

6 OpenCEM Testing

To verify the correct functioning of OpenCEM, it was tested in the real installation described in Section 4.7.

OpenCEM was active between July 23, 2023, and August 16, 2023, in the test installation. During the testing process, smaller bugs were identified and iteratively resolved.

6.1 Durability Test

OpenCEM ran in a durability test from August 10, 2023, to August 16, 2023. During this time, no interventions were made in the system. Logs of the devices and system events were recorded. OpenCEM was initialized via the CEM Cloud interface, through which the YAML configuration and SGr-XML files were downloaded.

The following section discusses the results of the durability test.

Figure 6.1 and 6.2 illustrate the relevant data from August 12, 2023. Figure 6.1 shows the consumption of the installed devices, the production of the PV system, and the room temperature (dashed line). The temperature scale is given on the right Y-axis. The variation of the consumption and production price of the used price controller (electric heater) is depicted in Figure 6.2.

August 12 was a day with changing cloud cover, which is evident in the fluctuating production. The electric heater has a price controller (20°C - 25°C temperature range), and the charging station has a dynamic surplus controller (1.5kW - 4kW).

During the first production peak around 10:00 AM, the electric heater is turned on. Subsequently, the production decreases, but the device remains active for the set 10-minute minimum runtime. This unintended reaction of the controllers to peaks could potentially be improved by filtering the production data.

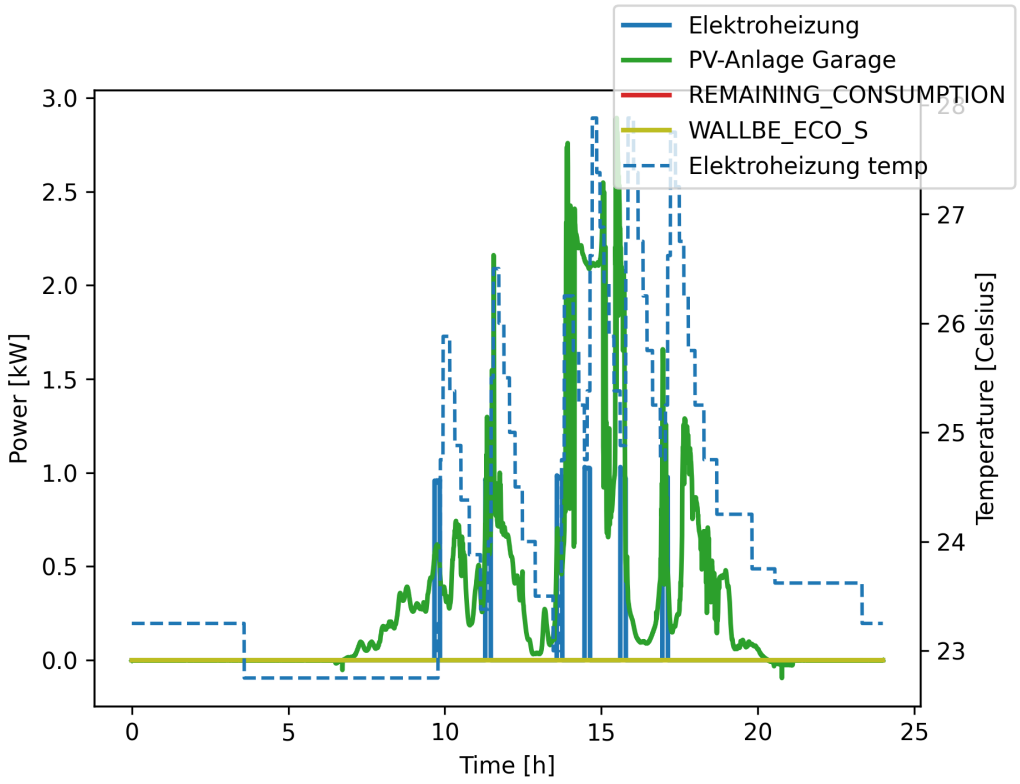

Figure 6.1: Data of the installation on a day with changing cloud cover.

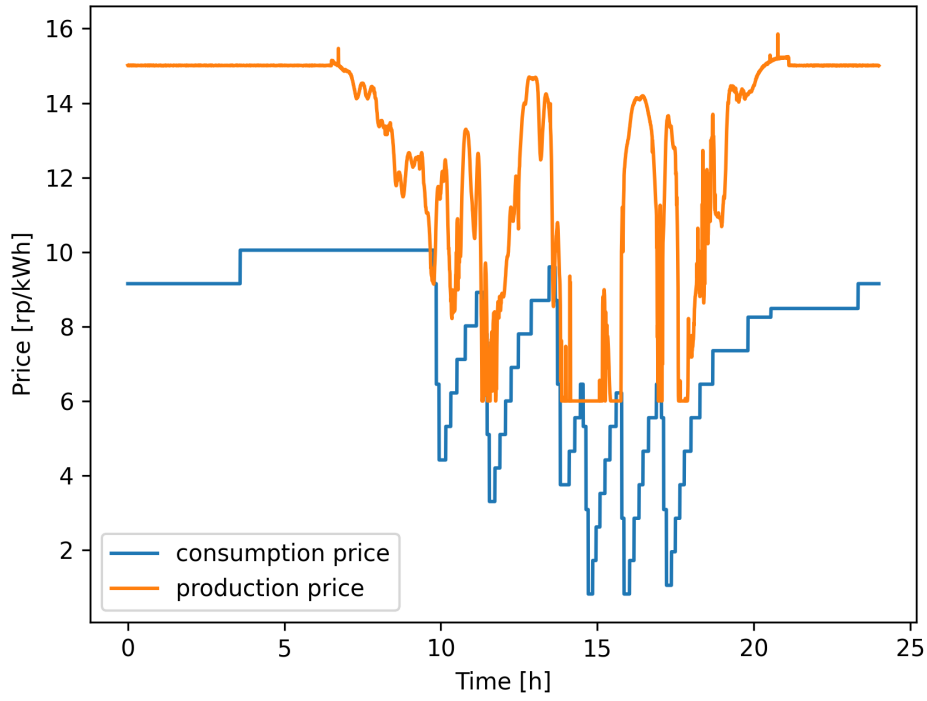

Figure 6.2: Temporal variation of production price and consumption price of the price controller of the electric heater

Figures 6.1 also show that the price controller responds correctly to the device's coverage and the room temperature. The controller is only activated when the actual temperature is below the maximum temperature. From Figure 6.2, it can be observed that the device is activated only when the consumption price is higher than the production price, as intended. Furthermore, the graphs reveal that the room temperature reacted rapidly to the activation of the electric heater. This was because the sensor was placed near the heater, and it was also activated when sunlight hit the PV system, leading to an increase in the room's temperature.

In Figure 6.3, the data from August 16, 2023, is presented. On this day, the vehicle was used by the client and was partially discharged. Around 11 AM, there was sufficient surplus to exceed the lower threshold of the dynamic surplus controller. Subsequently, the power consumption of the charging station adjusted to match the surplus. By 1 PM, the vehicle was fully charged, and the charging process was terminated.

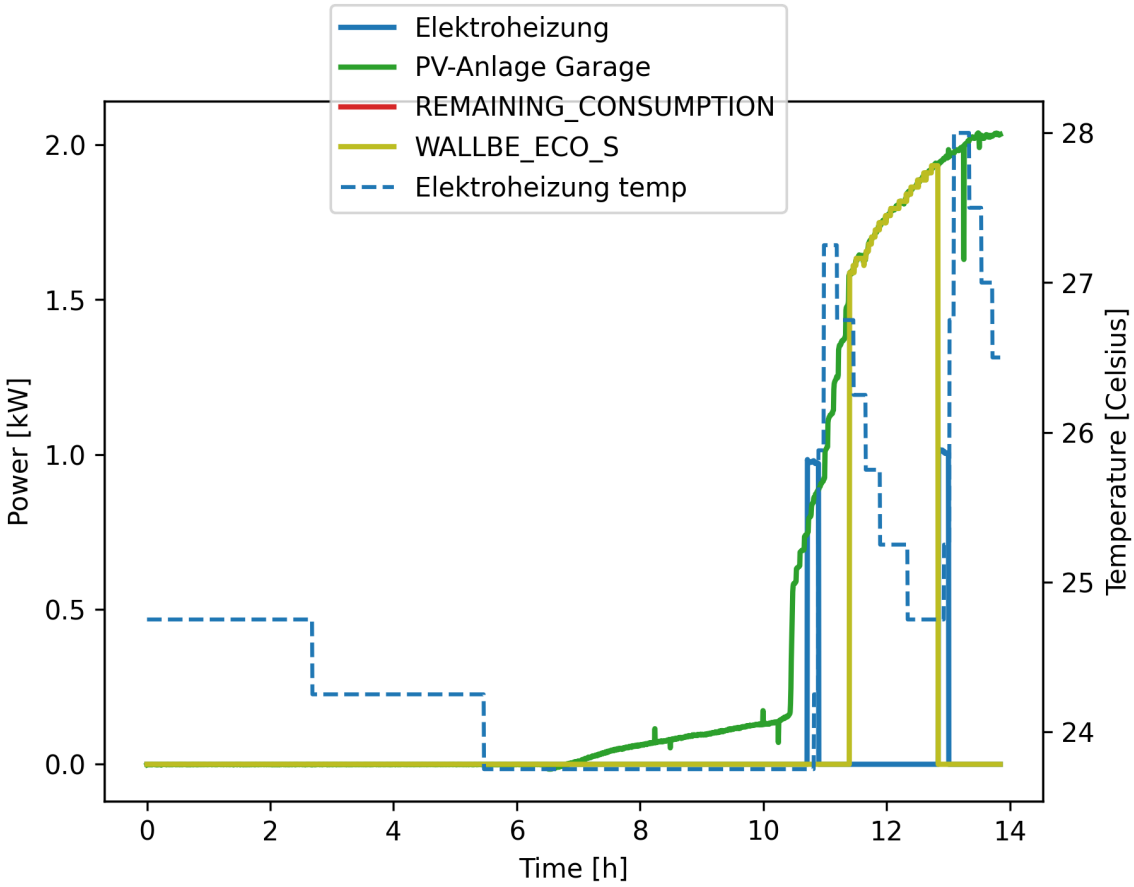

Figure 6.3: Data from August 16, 2023, illustrating the electric vehicle's charging process.

The durability test successfully confirmed that OpenCEM also functions flawlessly in a real installation. Communication with the devices was stable. Throughout the durability test, there was only a single short interruption. The used Shelly relay lost its Wi-Fi connection for one iteration. However, the error was caught by the error handling, as intended.

Quellenverzeichnis

[1] S. Ferreira, "P5 projektdokumentation", FHNW, Tech. Rep., 2023.

[2] *Ev charge control*, version c03, Phoenix Contact, 2014.

[3] *Installation und inbetriebnahme der ladesteuerung ev charge control anwenderhandbuch*, version 04, Phoenix Contact, 2020.

[4] *Aiohttp dokumentation*, Aug. 3, 2023. [Online]. Available: `https://docs.aiohttp.org/en/stable/` (visited on Aug. 3, 2023).

[5] "Uuid genertor". (Jul. 25, 2023), [Online]. Available: `https://www.uuidgenerator.net/` (visited on Jul. 25, 2023).

[6] A. Müller and N. Stutz, "Cloudbasierte Konfiguration eines Open-Source Energiemanagers, Projektbericht P5", FHNW, Tech. Rep., 2023.

[7] "Revpi unterstütze images". (Jul. 30, 2023), [Online]. Available: `https://revolutionpi.com/tutorials/images` (visited on Jul. 30, 2023).

[8] "Revpi-images installationsanleitung". (Jul. 30, 2023), [Online]. Available: `https://revolutionpi.com/tutorials/images/install-jessie` (visited on Jul. 30, 2023).

[9] "Advanced ip-scanner". (Jul. 31, 2023), [Online]. Available: `https://www.advanced-ip-scanner.com/de/` (visited on Jul. 31, 2023).

[10] "Putty downloadseite". (Jul. 31, 2023), [Online]. Available: `https://www.putty.org/` (visited on Jul. 31, 2023).

[11] "Realvnc installationsanleitung". (Jul. 31, 2023), [Online]. Available: `https://www.raspberrypi.com/documentation/computers/remote-access.html#virtual-network-computing-vnc` (visited on Jul. 31, 2023).

[12] "Python installationsanleitung für raspberry pi". (Jul. 31, 2023), [Online]. Available: `https://hub.tcno.co/pi/software/python-update/` (visited on Jul. 31, 2023).

[13] "Git installationsanleitung für raspberry pi". (Jul. 31, 2023), [Online]. Available: `https://github.com/cli/cli/blob/trunk/docs/install_linux.md` (visited on Jul. 31, 2023).

[14] "Github installationsanleitung für linux". (Jul. 31, 2023), [Online]. Available: `https://github.com/cli/cli/blob/trunk/docs/install_linux.md` (visited on Jul. 31, 2023).

[15] "Github authentifizierung". (Jul. 31, 2023), [Online]. Available: `https://docs.github.com/en/get-started/getting-started-with-git/caching-your-github-credentials-in-git` (visited on Jul. 31, 2023).