OpenCEM

Open Source Customer Energy Manager

Documentation V2.0.1

Fachhochschule Nordwestschweiz, Institut für Automation, Switzerland

in cooperation with SmartGridReady Association, Switzerland

Authors: Prof. Dr. David Zogg, S. Ferreira, Ch. Zeltner

October 2024

Version History:

Date	Version	Author	Changes
14.09.2022	0.1	D. Zogg	Document created
16.09.2022	0.2	D. Zogg	Document update, hardware components added
20.09.2022	0.3	D. Zogg	Chapter “Implementation of SmartGridReady” added
20.09.2022	0.4	S. Ferreira	Hardware priorities changes
04.10.2022	0.5	D. Zogg	Main program updated, EnOcean moved to Appendix
04.10.2022	0.6	S. Ferreira	Hardware priorities changes
07.10.2022	0.7	D. Zogg	Table of controllers added (Section 4.2)
08.11.2022	0.8	S. Ferreira	Hardware priorities changes
09.12.2022	0.9	D. Zogg	EnOcean components removed Hardware chapters translated to English and simplified
19.01.2023	1.0	S.Ferreira	Added chapters for implemented devices, logging and experimental setup
04.10.2024	2.0	D. Zogg	Document completely reworked according to simplified OpenCEM implementation
09.10.2024	2.0.1	D. Zogg	Controller with remaining power (priorisation)

Inhalt

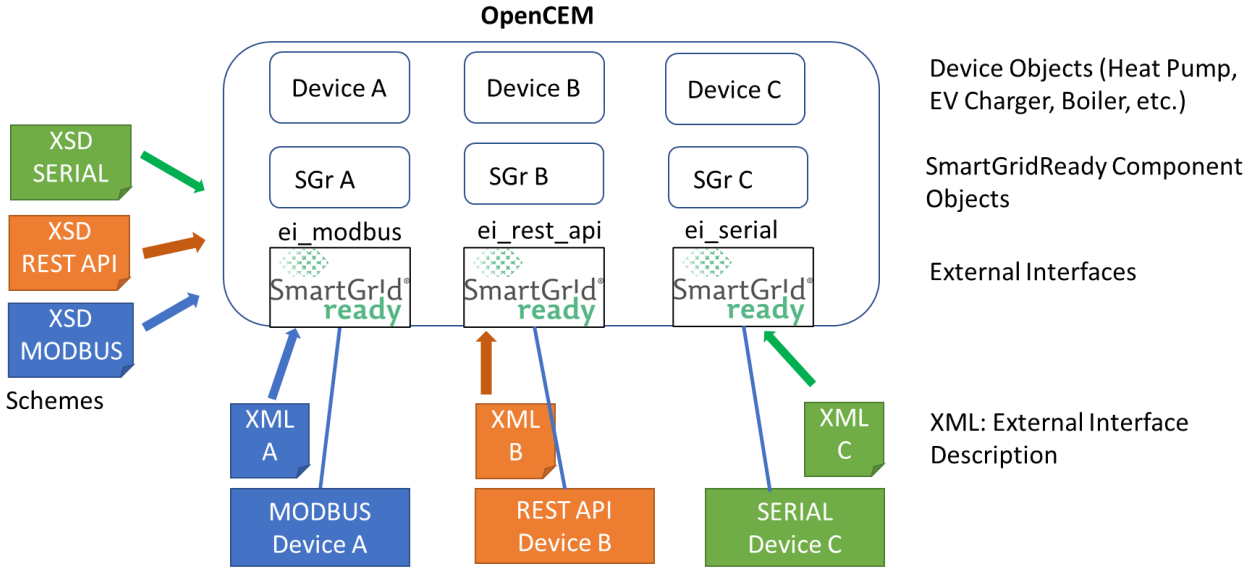
1 Purpose

The Open Customer Energy Manager (OpenCEM) demonstrates the functionality of the Smart-Gridready library as a prototype energy manager. The energy manager includes some example controllers required to control typical installations with pv plants, heat pumps, ev-chargers, and other devices. The controllers are tested in the lab of Fachhochschule Nordwestschweiz (FHNW).

The application may run on any PC or embedded device with Python 3.10 or higher installed. As development environment PyCharm is recommended from version 2022 or higher. The code either runs in simulation or hardware mode. Above, the devices may be configured “natively” or by the SmartGridready mechanism.

2 Implementation of SmartGridReady®

OpenCEM implements the core functionality of the SmartGridReady® mechanism, which enables standardized communication to all devices on the market in future. The implementation is shown in the following picture. The OpenCEM project is realized in Python and uses classes to define objects for each device, which is connected (e.g. device A may be a heat pump, device B an ev charger, device C a boiler, etc.). Each device object may contain a SmartGridReady component object, which connects the device to the SmartGridReady layer. The advantage of this approach is the independency of the main code from the SmartGridReady interfaces. The SmartGridReady layer implements the External Interfaces for each type of connection (e.g. MODBUS, REST API, SERIAL interface). The External Interfaces are automatically generated from XSD schemes. This generation is typically performed by the SmartGridReady® group in advance. At runtime, the corresponding XML files are read in for each device which is connected. The device is then automatically configured. As mentioned above, the OpenCEM project contains some example XML files which can be supplemented.



Implementation of the SmartGridReady® approach in OpenCEM
TODO: adapt this picture

3 Software Installation on PC for development

For development the platform “PyCharm” is suggested with integrated GitHub connectivity.

3.1 Repositories on GitHub

All required repositories are available on the SmartGridready Github Server:

Repository	Description
SGrSpecifications	SGrSpecifications defines the XML-schemes for the SmartGridready functional-profiles and the XML-schemes for device descriptions in XML, called EID-XML (External Interface Description XML).
SGrPython	SGrPython provides the source code of the communication handler library written in Python, called sgr-lib. This library may be installed by using “pip install sgr-lib”.
SGrPythonSamples	SGrPythonSamples contains the OpenCEM code as an example application.

3.2 File Structure of OpenCEM

The OpenCEM project in SGrPythonSamples contains the following files and folders:

Folder/file	Description
OpenCEM/ cem_lib_components.py	This library contains all device classes for OpenCEM
OpenCEM/ cem_lib_controllers.py	This library contains all controller classes for OpenCEM
OpenCEM/ cem_lib_loggers.py	This library contains the logging functionality for OpenCEM
OpenCEM/ cem_lib_auxiliary_functions.py	This library contains the auxiliary functionality for OpenCEM such as web functions and YAML parsing
OpenCEM/native_device.py	
templates/ index.html	Here the HTML code for the local web page is stored.
xml_files/ SGr*.xml	Here a copy of the actual SmartGridready EID-XML files are stored, with which the OpenCEM code has been tested. Note that new versions of XML files must be manually copied from SGrSpecifications.
yaml_files/	
yaml/ openCEM_settings.yaml	This file contains the general settings for OpenCEM. It also contains the name of the actual configuration file (see below).
yaml/ openCEM_configuration.yaml	This file contains the actual configuration of OpenCEM, which includes the devices, controllers and communication channels.
GUI_Server.py	This file implements the web server for the local GUI (graphical user interface)
OpenCEM_main.py	This file implements the **main program** of OpenCEM. Run this file in order to start OpenCEM.
requirements.txt	This file contains all required libraries for OpenCEM together with the appropriate versions. PyCharm uses this file to install the Python Packages. TODO: check this!

3.3 Installation on Development Environment

Run the following steps to install OpenCEM source:

- Install Python 3.10 or later
- Install PyCharm 2022 or later
- Download all files above from GitHub repository SGrPythonSamples
- Open PyCharm and install required Python Packages
- Install the SGrLibrary using “pip install sgr-lib”

3.4 Configuration of OpenCEM

OpenCEM is configured by some files, which are read at startup of the OpenCEM_main. These files are described in the following sections.

3.4.1 Settings File (OpenCEM_settings.yaml)

This file contains the general settings for OpenCEM. The name of the file may not be changed.

```yaml
# general settings for OpenCEM

loop_time: 300  # time for the loops in seconds
simulation_speed_up: 100 # put 1 here if system is not simulated
duration: 0 # in seconds (0 means will run forever)

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

# local path for configuration
path_OpenCEM_config: "yaml/openCEM_config.yaml"

# credentials for loading configuration from cloud
backend_url: "" # leave empty if no cloud
installation: ""
token: ""

# local web server
ip_address: "192.168.0.76"
port: "8000"
```

TODO: automatic detection of local web server ip_address.

3.4.2 Configuration File (OpenCEM_config.yaml)

This file contains the lists of devices, controllers and communication channels for the running of OpenCEM on a specific hardware configuration. The devices may also be simulated.

Example configuration for simulation:

```yaml
# actual device and controller configuration of OpenCEM

installationName: SimTest
creationTimestamp: 2024-09-26 08:55:00
version: 1
communicationChannels:
- name: MODBUS_RTU
  type: MODBUS_RTU
  extra:
    port: 502
    baudrate: 9600
    parity: E
    stopbits: 1
    bytesize: 8
    timeout: 1
- name: REST_API_CLEMAP
  type: REST_API
  extra:
    baseURL: https://cloud.clemap.com:3032
    username: smartgridready24@gmail.com
    password: *****
- name: REST_API_SMARTME
  type: REST_API
  extra:
    baseURL: https://api.smart-me.com
    username: smartgridready2024@gmail.com
    password: *****
- name: Shelly_Cloud
  type: REST_API
  extra:
    baseURL: https://shelly-113-eu.shelly.cloud
```

```
    shelly_cloud_key:
MjVmZTY3dWlk1C6347B059038FDE0A94611014FADBB0908F5A53A5C312DF813E8ECEDAB9D4ABC
BB10214E6FC7AB2
devices:
- name: Shelly pro3EM
  type: POWER_SENSOR
  smartGridreadyEID: None #SGr_04_mmmm_dddd_ShellyPro3EMEIV0.2.1
  EID_param: None #config_Shelly_Pro3EM.yaml
  nativeEID: Shelly_Pro3EM_local.yaml
  simulationModel: None
  isLogging: true
  communicationChannel: None
  extra:
    device_id: "0"
    baseURL: http://192.168.137.80
- name: CLEMAP EMS
  type: POWER_SENSOR
  smartGridreadyEID: None
  EID_param: None
  nativeEID: CLEMAP.yaml
  simulationModel: None
  isLogging: true
  communicationChannel: REST_API_CLEMAP
  extra:
      sensor_id: 661d109a441239001393c201
- name: Hoval
  type: HEAT_PUMP
  smartGridreadyEID: None #SGr_04_mmmm_dddd_ShellyPro3EMEIV0.2.1
  EID_param: None #config_Shelly_Pro3EM.yaml
  nativeEID: HOVAL.yaml
  simulationModel: None
  isLogging: true
  communicationChannel: MODBUS_HOVAL
  extra:
    ipaddress: 192.168.137.246
```

TODO: Add example configuration for hardware

Contents of the configuration file

The configuration file includes a list of devices to be managed by OpenCEM. These devices can range from sensors and actuators to more complex systems such as heat pumps or electric vehicle chargers. Additionally, the file specifies a list of communication channels.

Devices Properties:

Property	Description
name	Name of the device. Must be a unique string. The devices are referenced by name.
type	Type of the device. The following types are supported: “POWER_SENSOR” – a sensor device which measures the electric power (and energy) “TEMPERATURE_SENSOR” – a sensor device which measures the temperature (for room or storage temperature) “RELAIS_SWITCH” – an actuator device which switches a relais on or off (may have more than one switching channel) “EV_CHARGER” – a device representing a charging station for electric vehicles (with variable charging power) “HEAT_PUMP” – a device representing a heat pump for room heating and domestic hot water production
smartGridreadyEID	File name of the EID-XML (SGr*.xml). If this file name is set, the SmartGridready functionality is enabled for the current device. If no EID-XML is existing, this property may be set to “null”.
nativeEID	File name of the “native” YAML device description (Device*.yaml). If this file name is set, the native functionality is enabled for the current device. If no native YAML is existing, this property may be set to “null”.
isLogging	If this boolean is set to “true”, the device logger is activated for the specific device. Else set to “false”.
communicationChannel	Name of the corresponding communicationChannel (see table below). The communication channel defines the hardware channel through which the device communicates. For simulation set it to “null”.
extra	Defines extra settings for the device. According to the device type, the following entries may be set: “address” – IP or other address of the device “port” – IP port of the device “nChannels” – number of channels (for relais actuator) “minPower” – minimal power consumption of the device in kW (kilowatts) “maxPower” – maximal power consumption of the device in kW (kilowatts) “nominalPower” – nominal power consumption of the device in kW (kilowatts) “phases” – number of phases (for EV charging station)

The configuration file also contains a list of **controllers**, with which the devices are controlled by OpenCEM. Controllers may be simple switching controllers to more sophisticated implementations. Here the focus is set on generic solutions for basic optimization of self-consumption for buildings with photovoltaic plants (PV).

Note *that the controllers are only tested in lab conditions, and not in real applications. For example, the controllers have no time delay for switching. In real applications frequent switching of devices may damage the devices (especially heat pumps). Thus, be careful when using for real world applications. Always test and optimize the controllers for your application!*

TODO: extend with controllers for grid optimization such as dynamic price controllers etc.

Communication Channel Properties:

Property	Description
name	Name of the controller. Must be a unique string. The controllers are referenced by name.
type	Type of the controller. The following types are supported: “SWITCHING_EXCESS_CONTROLLER” – switch on/off according to actual PV excess. Used to control simple household devices or boilers. “DYNAMIC_EXCESS_CONTROLLER” – variable setpoint according to actual PV excess. Used to control EV chargers, “smart heaters” or other variable controllable devices. “TEMPERATURE_EXCESS_CONTROLLER” – variable temperature setpoint according to actual PV excess. Used to control heat pumps, boilers or other devices, which provide a variable temperature setpoint. The PV excess is calculated by the power production of the PV plant, subtracted by the overall consumption of the building. Note that the own consumption of the device is considered to prevent constant switching. The temperature excess controller rises the temperature setpoint according to the PV excess.

TODO: Describe communication channels and give a table for their properties (like above).

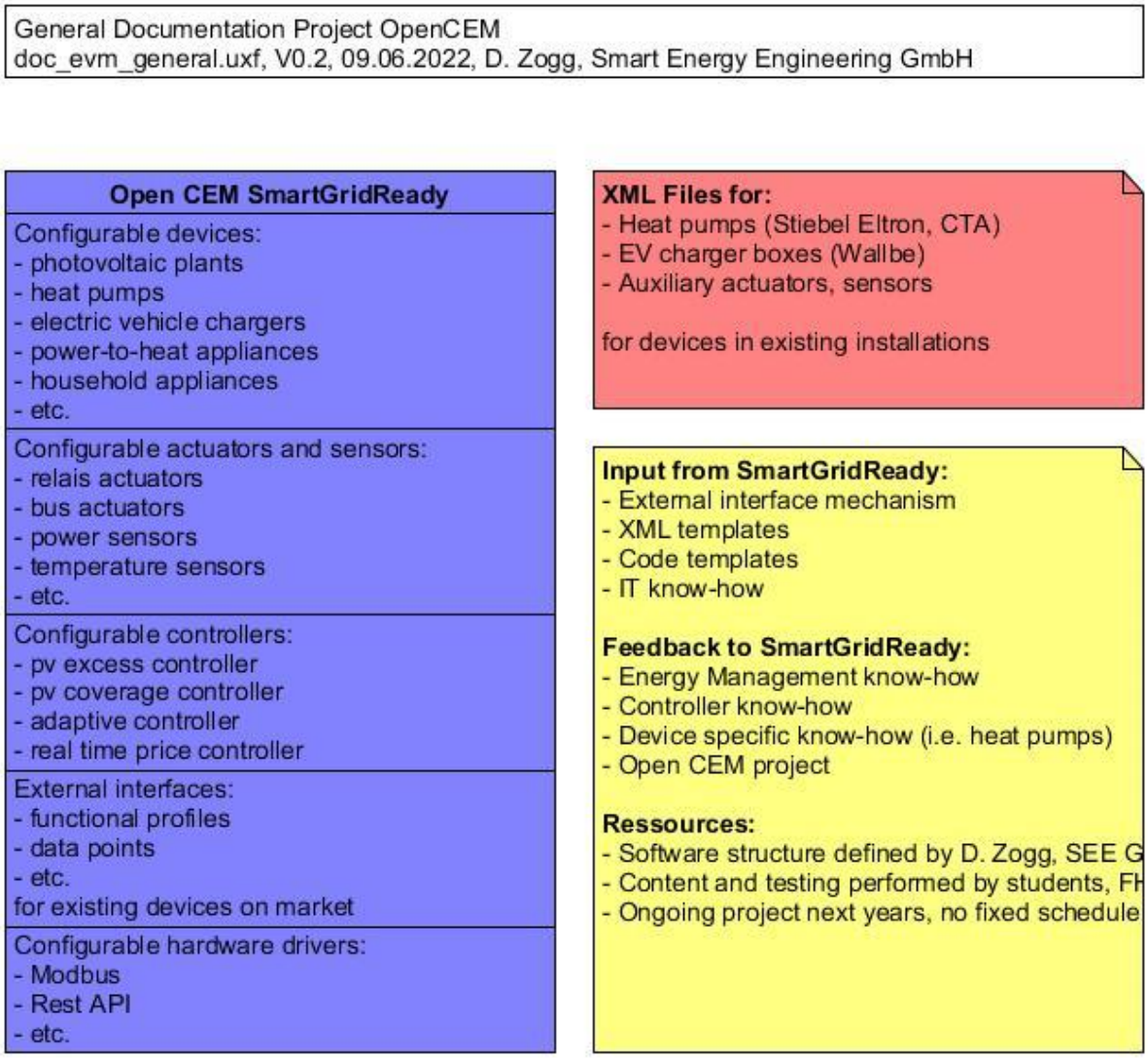
4 Software Structure

In this chapter the software structure of OpenCEM is described. The focus is set on the functional part of the application on the higher layers. For a full documentation of the underlying SmartGridReady functionality, refer to the documentation of SmartGridReady.

4.1 General Overview

The following diagram gives an overview of the OpenCEM software structure.

TODO: Adapt this diagram and the text below!



(Reference: Documentation/doc_evm_general.jpg)

A short description will be given starting from the top layer to the bottom layer (left side):

On the highest level of the software, the devices of an installation are configured. This includes photovoltaic (pv) plants, heat pumps, electric vehicle (ev) chargers, etc. Any number and type of devices may be controlled by the software. On the next level the actuators and sensors are defined. Actuators may be simple relay contacts or more sophisticated actuators running over digital bus systems. On the other side sensors may be defined. Here the most comm types are power meters or temperature sensors, but heat sensors or any other type may be added. The number of sensors is not limited. The sensors and actuators are typically connected to devices. Thus, a heat pump may have a relay actuator and a several temperature sensors.

On the next level, the controllers are defined. Here the most common controllers are implemented which are used in optimization of photovoltaic self-consumption (“Eigenverbrauchsoptimierung”), including pv excess and coverage controllers. Above, an innovative real-time price controller is implemented, which can be used to control large thermal masses such as buildings or future applications with varying tariffs. The controllers are explained more detailed in Chapter 4.3.

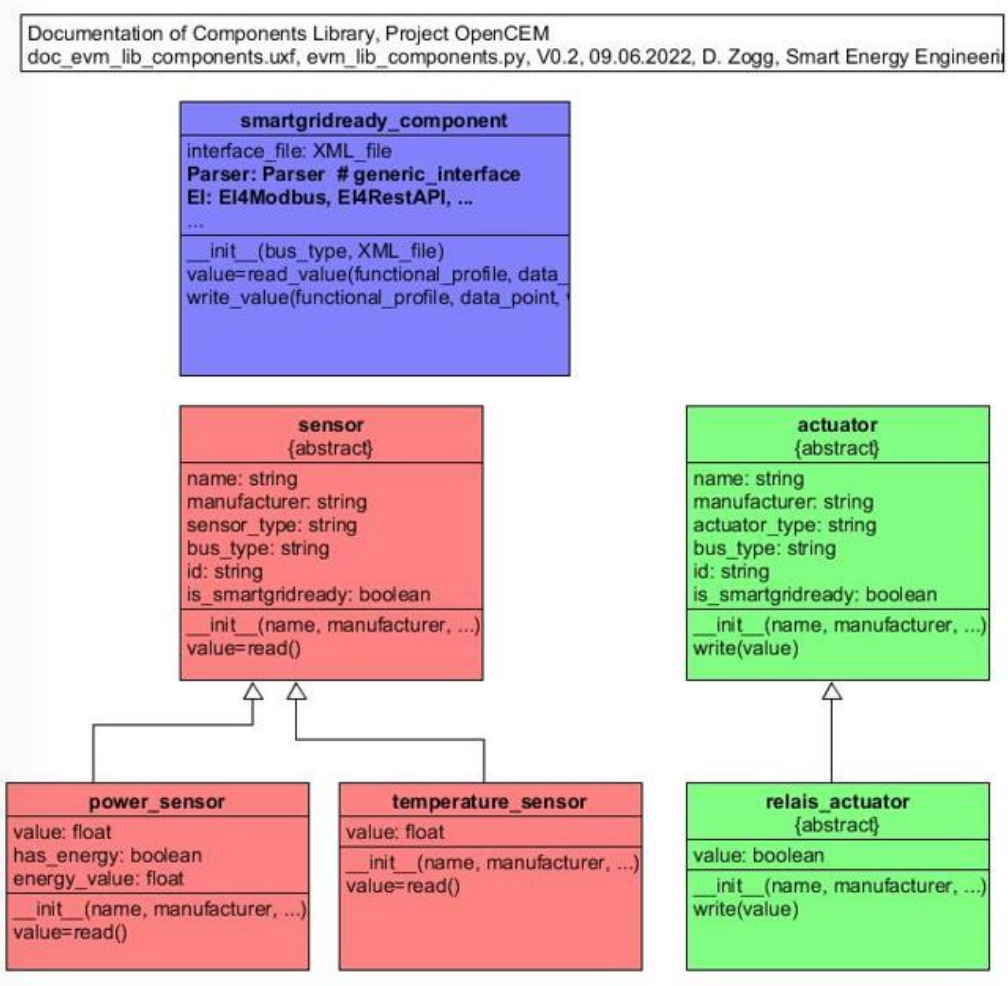
On the lower levels, the generic hardware communication is implemented using the mechanism of SmartGridReady. On the “External Inferface” level, the functional profiles and data points are described for existing devices on the market. For that the XML files are used (see Chapter 0), which are read in for a specific device. The application uses a generic interface to communicate with device, which makes it independent of manufacturer specific implementations. The specific interfaces are on the lowest level. They are part of the SmartGridReady package and must not be implemented by the CEM programmer. There are hardware drivers for the mostly spread interfaces such as MODBUS, REST API, serial interfaces, relay contacts, etc.

The SmartGridReady mechanism will shortly be described with an example of a heat pump: These devices are typically controlled by MODBUS communication, which defines a specific register address for each data point such a setpoint or a measured temperature. The problem is, that the register addresses as well as the number of registers and even their content vary from one manufacturer to another. Thus the programmer of the CEM has to implement a huge variety of solutions. This is simplified very much with SmartGridReady. Here the programmer only needs to implement *one* driver for all heat pump types using a generic interface. For that he selects a pre-defined functional profile of the heat pump, for example a “base profile” with which he can switch the device to several states (using a generic data point “device state”). The names of these states are standardized as well (e.g. “off”, “on”, “standby”, “eco”, “comfort”, etc.). On the other side, he may select a profile “buffer storage” and read the data point “temperature” from it. The heat pump then delivers the temperature of the heating buffer storage tank if available. For a proper working of the mechanism, the manufacturer of the heat pump simply has to deliver an appropriate XML file (which later can be read in from a SmarGridReady database by simply scanning a QR code on the label of the device). This mechanism simplifies interoperability in a significant way. Especially, the work for the CEM integrator is significantly reduced. Of course, the consistency of the XML files need to be tested before at real applications. This is the aim of the OpenCEM project. The more devices are tested in real applications, the more robust the mechanism will work. You are part of the future!

4.2 Devices / Components

TODO: Adapt this diagram and the text below!
There are only device classes, no sensor and actuator classes anymore!

The components are implemented in the module *evm_lib_components*. Components are either devices like heat pumps, boilers etc. or actuators and sensors. Actuators and sensors may also be connected to devices, e.g. for controlling a heat pump device by a relay actuator. The following UML charts give an overview (separated into two parts).



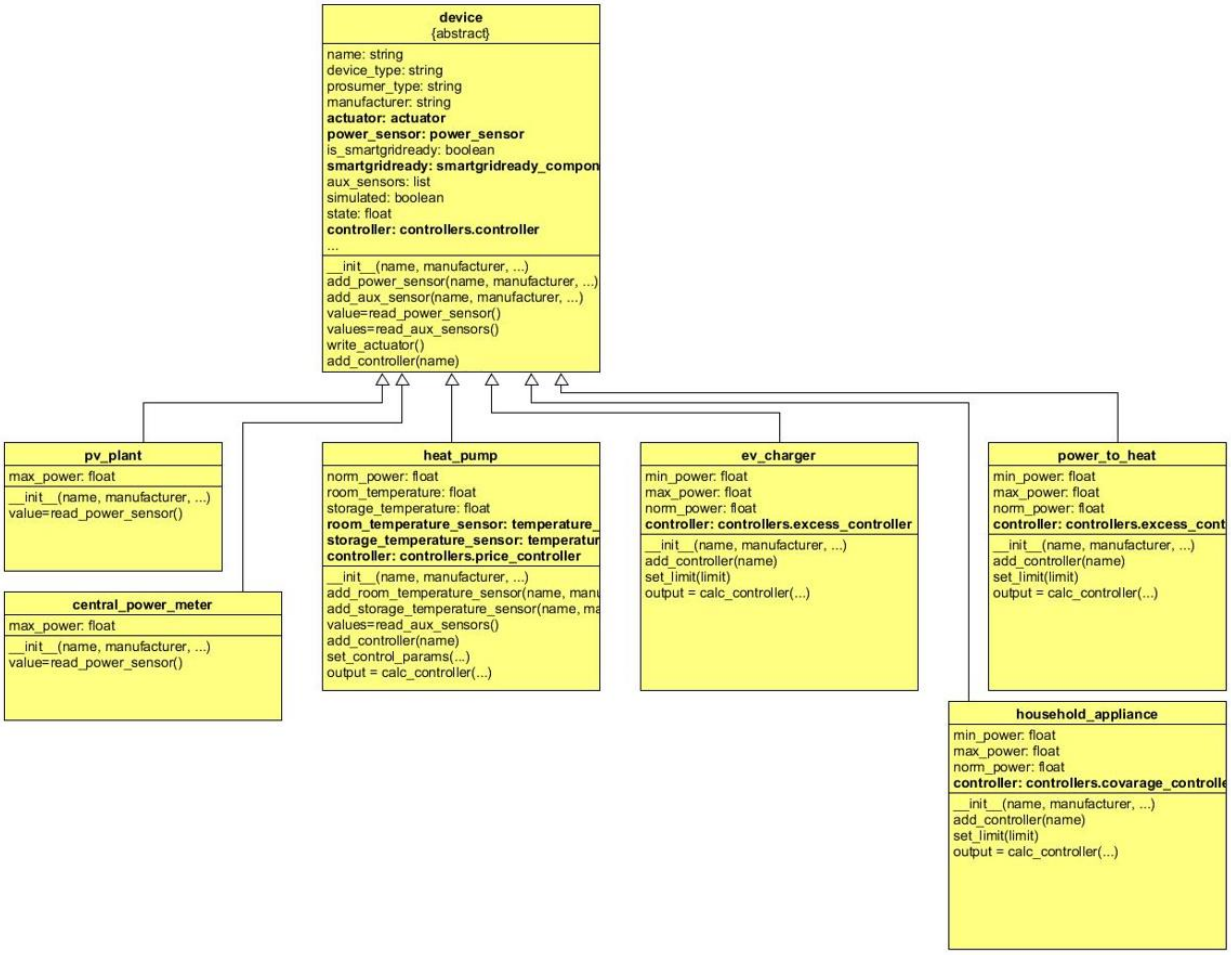
(Reference: Documentation/doc_evm_lib_components, part I)

The following **component** classes are available:

- **smartgridready_component**: Defines a component with SmartGridReady functionality. For that the corresponding XML file name is the most important property, which connects the component to the external interface. Depending on the bus type, an appropriate interface reader is selected (e.g. MODBUS, REST API, etc.). The *read_value* and *write_value* methods use the underlying SmartGridReady *get_val* and *set_val* methods for generic access of the device. As parameters, the functional profiles and data points defined in the connected XML file are used. In fact, this class is a “wrapper” class which ensures compatibility to the actual SmartGridReady interface (also when this might change in future versions).
- **sensor**: This is an abstract class with common properties and methods for sensors. It is not directly used in the application.
- **power_sensor**: Represents a power sensor, e.g. a smart meter which measures the actual consumption power of another device or the complete household. The measured value is the power in kW (kilowatt) and may be positive or negative (for bidirectional sensors). Optionally the sensor may deliver an energy value. Energy meters usually have at minimum one register which summarizes the consumed (or produced) energy in kWh

(kilowatt-hours). For bidirectional meters, the energy value is separated in import and export (two registers). Even if on the market meters with more than two registers exist (e.g. 4 registers for different tariffs and import/export), they are always mapped to one or two registers.

- **temperature_sensors**: Represents a temperature sensor. The measured value is the temperature in °C (degree Celsius). On the market, various temperature sensors exist, e.g. cabled sensors for measuring storage tank temperatures, wireless sensors for measuring room temperature, etc. All these sensors are represented by the same class.
- **actuator**: This is an abstract class with common properties and methods for actuators. It is not directly used in the application.
- **relais-actuator**: Represents a simple switching relay, which has two state “on” an “off” (boolean value “true” or “false”). If a device is switched by more than one relay input (e.g. heat pump with SG-Ready interface), two relais-actuators may be defined.



(Reference: Documentation/doc_evm_lib_components, part I)

The following component classes are available for **devices**:

- **device**: Defines an abstract class for a generic device. A device may have an actuator and/or sensors connected. If the actuator is set to “None”, no (external) actuator is connected, then the device must be controlled by an internal signal (e.g. over MODBUS). The same is true for the sensors. In general a device is measured by an (external) power sensor. But it can also be “measured” by an internal signal. Auxiliary sensors may be connected (there is a list of connected sensors). Additionally, each device can have SmartGridReady functionality by connecting to a smartgridready_component. And each device contains a controller object (see Chapter 4.2). Above, each device may be simulated or connected to hardware. In the simulation mode (simulated = true), the sensor values are internally calculated from the actuator values by a simple model. The logic of the model depends on the derived class and may be changed by the programmer. With

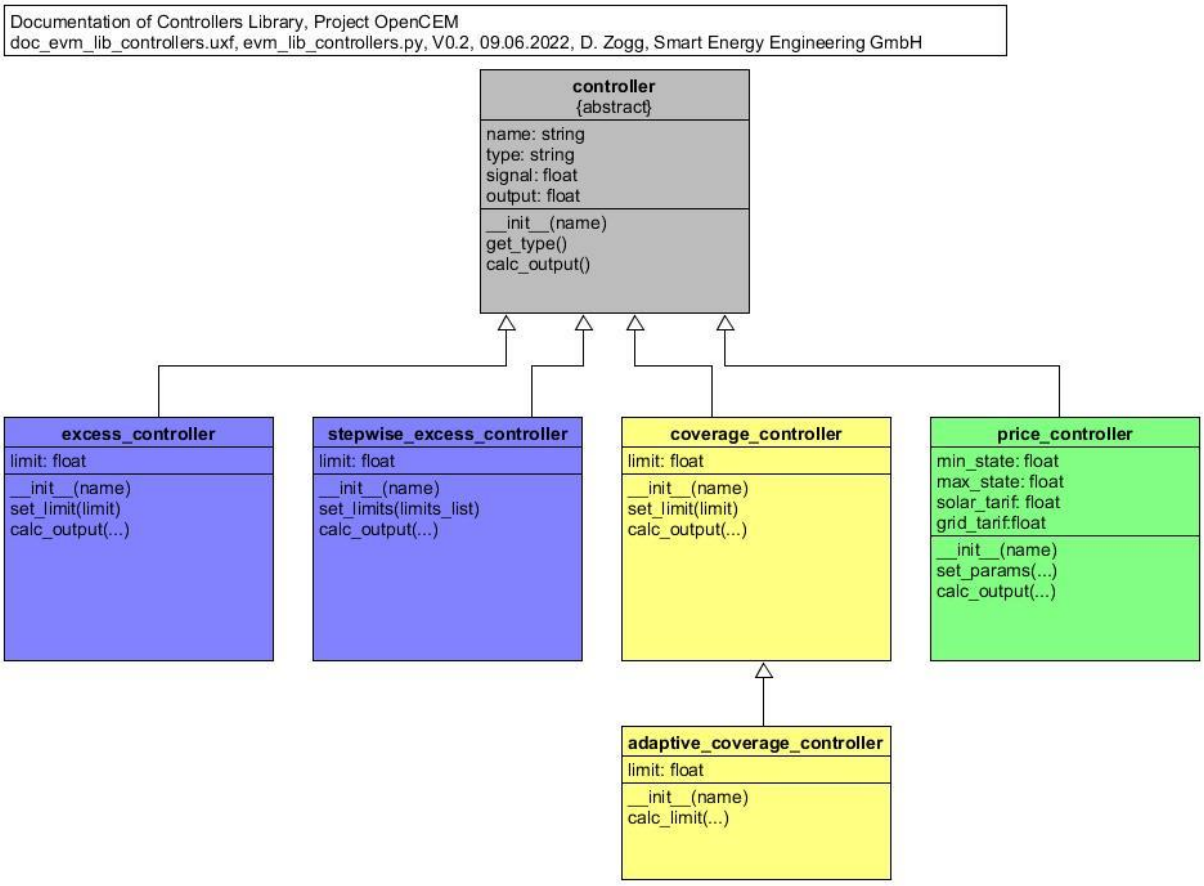
this feature, the logic of the controllers and entire system can be tested in a software loop before going to hardware, which speeds up development. Also a co-simulation is possible with some devices as real hardware and other devices simulated. The generic device is not used in the application, but the following classes below.

- **heat_pump**: This device has special functionality for heat pumps. The nominal power consumption (in kW) is defined as a property (which must be set from a manufacturer data sheet). Additionally, a room temperature and a storage tank temperature are measured values, used for control. Therefore an (external) room temperature and/or storage temperature sensor may be connected. Alternatively, the values can be read internally from the heat pump, if no sensors are connected. As a controller, a price controller is connected by default. This type is best suited to the control of whole buildings with room temperature measured.
- **ev_charger:** This device has special functionality for electric vehicle chargers. Here a minimum and maximum power consumption is defined (in kW). This range is used for power control. As default, an excess controller is connected which directly sends the value of the pv excess in kW to the charger (or calculated to A=ampere). Below the minimum power, the charger will be switched off (usually at 6A according to today's AC type charging norm). The charger feeds back the charging state of the vehicle, e.g. "A" for ready, "B" for connected, "C" for charging, etc. Note that with the actual AC charging norm the SOC (stat of charge) of the internal car battery cannot be read out. Future DC charging norms will deliver this important value (CCS, CHAdeMO). Also bidirectional charging will be possible. For that purpose, the code of the excess controller must be adapted.
- **power_to_heat**: Represents a simple power-to-heat application such as an electric heating element in a boiler, including "smart pv appliances" which can be controlled continuously. Therefore, the excess controller is connected by default, which controls between a defined minimum and maximum power limit. They may also be controlled in an on/off mode by setting a power limit. *Note that direct power-to-heat applications are inefficient and should be avoided or replaced by heat pumps, especially in winter time when there is low pv production!*
- **Household_appliance**: Represents a household appliance such as a washing machine, dishwasher, dryer, air conditioner, pool heater, etc. Usually these appliances can only be controlled by switching them on or off. Therefore a coverage controller is used. *Note that the proper working of the appliances must be tested in practice, some devices do not start up themselves after switching power off.*
- **pv_plant:** This simple device represents the photovoltaic plant and cannot be controlled. It has a power sensor connected, which delivers the actual pv production (in kW). Alternatively, the value can be read out of the AC inverter (no external sensor required).
- **central_power_meter:** This simple device represents the central power meter which is typically installed at the grid connection point in order to measure the power import and export of the whole building. It always has a power meter connected, optionally with energy values measured for import/export.
- **battery_power_meter** (not yet implemented): If a battery is installed, the import and export to the battery also has to be measured. Therefore a bidirectional power meter is installed (for AC coupled batteries only) or the power value is read out of the battery inverter.

4.3 Controllers

The controllers are implemented in the module *evm_lib_controllers*. The following UML chart gives an overview.

TODO: Adapt this diagram and the text below!
There are only excess controllers so far, more sophisticated controllers may be developed.



(Reference: Documentation/doc_evm_lib_controllers)

The following controller classes are available:

- **controller** (“Regler”): Abstract base class with common properties and methods. Can not be used in the code directly.
- **excess controller** (“Überschussregler”): Controls the power production of the photovoltaic plant less the total power consumption of the household, also known as “pv excess”, measured in kW (kilowatts). The device is set to the resulting power value (kW). For example, an ev charger or a simple “power-to-heat” application is controlled by this algorithm. Also, a limit can be defined for an on/off switching device. For example, an electric heater can be switched on and off at a given level of kW. Also a combination of switching and continuous output is possible, e.g. for ev chargers. Typically, there is a minimum power limit, below which the ev charger must be switched off (in order to reduce charging losses).
- **stepwise excess controller** (“stufenweiser Überschussregler”): Works like the excess controller, but only outputs discrete states instead of a continuous signal. For example, heat pumps with the “old” SG-Ready interface are controlled by four states, such as 0 for “off”, 1 for “normal operation”, 2 for “desired operation” and 3 for “forced operation”. Another example are electrical resistance heaters for boilers which can be switched in different stages. These controllers have more than one switching limit, namely one limit for each state.
- **coverage controller** (“Deckungsgradregler”): Controls the solar coverage of a device. The solar coverage is the percentage of the consumption of the device, which is covered by pv excess (100% fully covered by pv, 0% not covered at all, 50% half covered by pv, half by the grid). The advantage of this controller is its independency of the scale of the device. The controller may be used to switch on or off a simple household device or even a heat pump over a relay switch. The base coverage controller works with a fixed switching limit (which can be set to 100% for having full solar coverage or to a lower value, e.g for heat pumps, to give them a chance to switch on in winter).

- **adaptive coverage controller** (“adaptiver Deckungsgradregler»): Works like the coverage controller, but with an adaptive limit. The limit is calculated from the expected maximum pv power of the day and the typical power consumption of the device. The advantage of this controller is the adaptation to the season. It may also be used for heat pumps or other devices, which must run at low pv power in winter time and high pv power at summer time.
- **real time price controller** (“Echtzeit-Preisregler”): This controller reacts on a varying price of the (local) electricity. It automatically finds the minimum price for running the device. In fact, two price signals are calculated in real-time: an offering price depending on the actual ratio of pv production an grid consumption, and an asking price depending on the state of the device. For heat pumps or thermal appliances, the state of the device is the temperature. For controlling whole buildings, the room temperature is measured as the state. This controller is very useful, since it might also consider varying prices coming from the grid (in future).

For a detailed description of the controller classes, their properties and methods, see the comments in the source code (module *evm_lib_controllers.py*).

5 Installation on Embedded System for deployment

TODO: Test OpenCEM on KUNBUS in lab and adapt this chapter!

5.1 Controller

Used hardware: KUNBUS RevPi Flat

Note: The RevPi Flat has no possibility to directly access the internal memory. For that it is safe from manipulations. But as a disadvantage the operating system can not be recovered at damage. Be careful with “super user commands” and similar actions!

Description of the interfaces:
https://revolutionpi.de/tutorials/uebersicht-revpi-flat/

5.2 Setup of the Controller and Development Tools

The setup of the RevPi is described here:
https://revolutionpi.de/tutorials/quick-start-guide-revpi-flat/

Connect the device to the local network (which must have Internet access). Use the Home Router or any IP scanning tool to find the IP address of the RevPi. The following tutorial helps:
https://revolutionpi.de/tutorials/uebersicht-revpi-flat/revpi-flat-ohne-separaten-bildschirm-tastatur/

The device has an integrated web server for simple configuration:
https://revolutionpi.de/tutorials/software/webstatus-fuer-stretch-flat/

For logging in, the default user and password are printed on the label of the device.
For the web server, use “admin”.

Install Remote SSH Connection for Visual Studio Code:

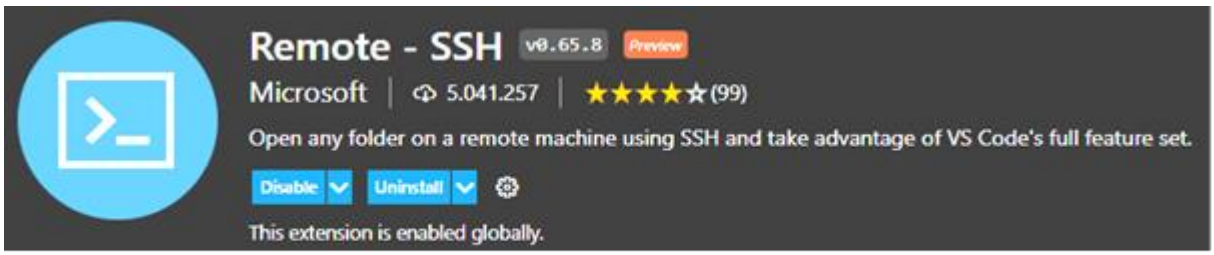
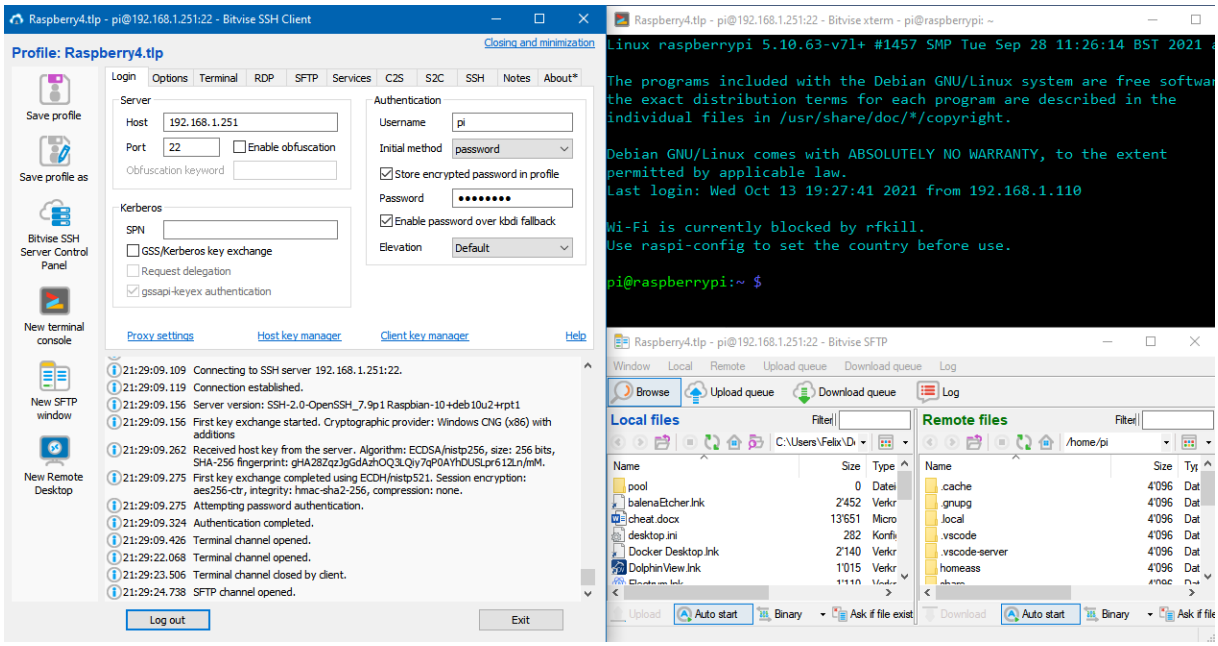


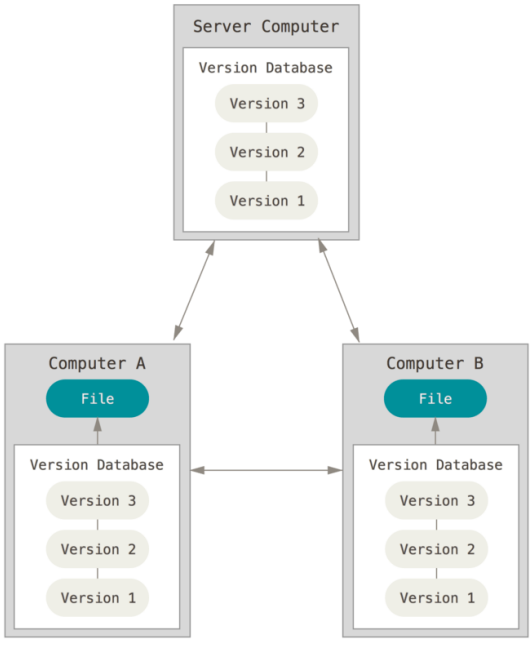
Abbildung 1: Remote SSH im VSCode Store

Install Bitvise SSH Client for up- and downloading code and external terminal:



For SSH, the default user and password are printed on the label of the device.
Use “pi” as username. Fill in the IP address of the device at “Host” and set “Port” to 22.

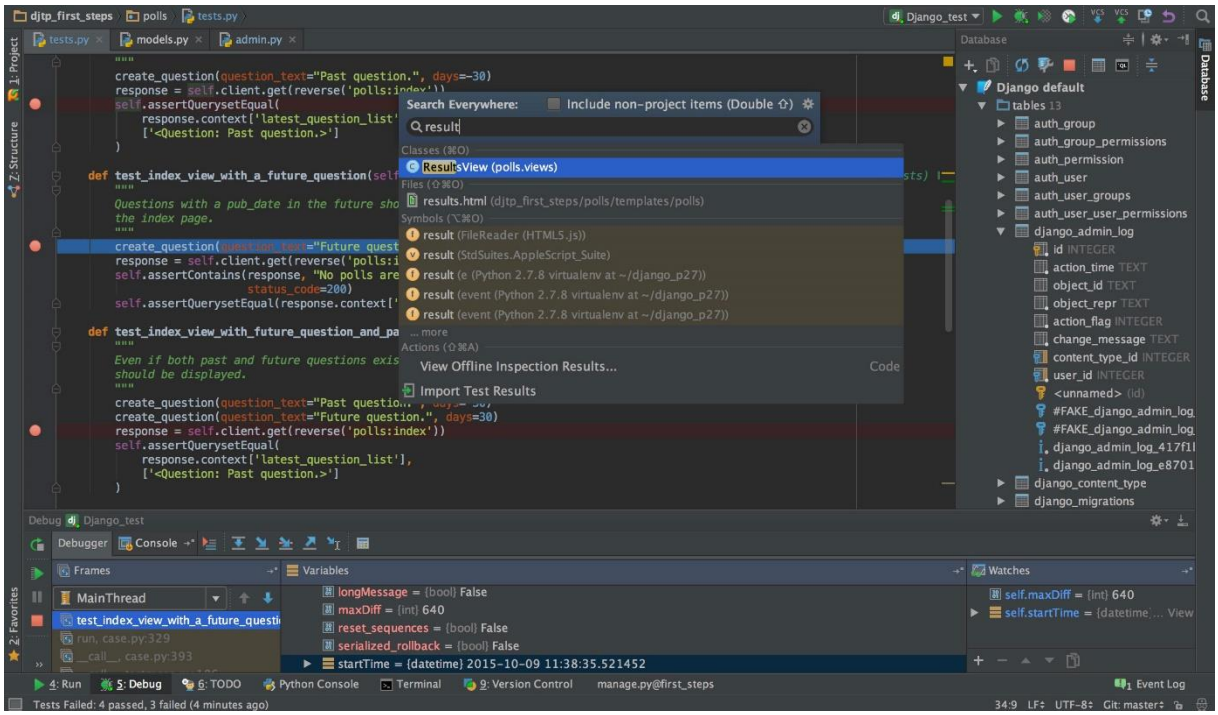
Install a version control system such as GitHub (or FHNW GitLab)



Computer A may be the development PC, Computer B the embedded target (RevPi). This way a simple updating and versioning of the controller software is possible.

A “master” branch is used for the main version of the software. New “branches” are created for new developments. They are merged with the “master” branch, when the development is finished (or the code is tested and verified).

Install PyCharm IDE by JetBrain® for easy code development and debugging:



Integrate GitHub to PyCharm for versioning and updating the controller.

Install Python 3.8 (or higher) if not already installed:
https://www.python.org/downloads/release/python-380/

Update all libraries. Also refer to Chapter **Fehler! Verweisquelle konnte nicht gefunden werden.** for a list of the required libraries.

First use the development PC for testing and debugging the code. For that connect all external hardware to the PC (see chapter 0). Then download the code to the target.

Use “git pull” on the target to download the code from the GitHub repository.

On the target, the main program must be run as a “systemd” service, which is automatically started at the booting of the device:

- Use “systemctl restart CEM.service” to manually restart the service (e.g. after downloading a new software version).
- Use “systemctl status CEM.service" to see the status of the service (it must be running)
- Use "sudo nano /etc/systemd/system/CEM.service" to edit the service script if necessary

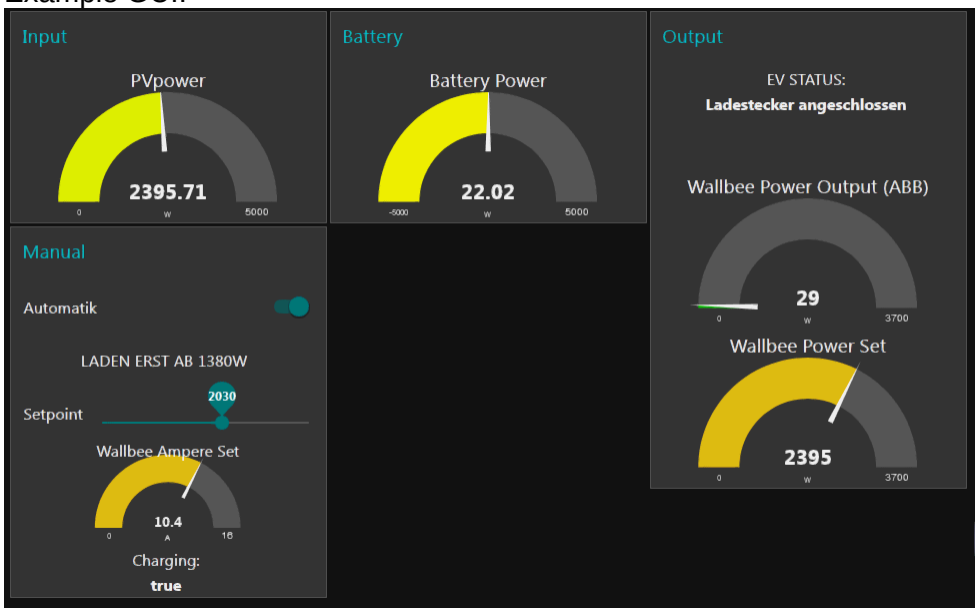
Use the “Remote Debugging” ability in PyCharm to debug the code on the embedded controller directly (if supported!).

5.3 User Interface

Node-RED

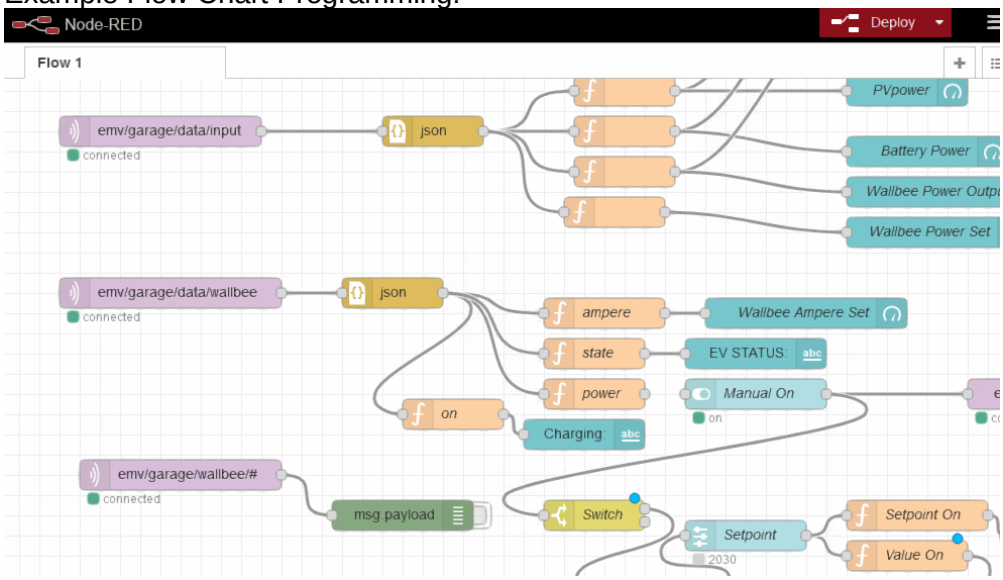
A simple user interface is realized by Node-RED, which is already installed on the RevPi.

Example GUI:



(Reference: Bachelor Thesis F. Bögli FHNW)

Example Flow Chart Programming:



(Reference: Bachelor Thesis F. Bögli FHNW)

The Node Red User interface can be easily accessed by the web browser over the local IP of the device at port 1880:
192.168.xxx.yyy:1880/ui → Graphical User Interface
192.168.xxx.yyy:1880 → Flow Char t

If the user interface contains more than one page, use the menu or the following commands to navigate:
http://192.168.1.224:1880/ui/#!/0 → navigate to page 0 (main GUI)
http://192.168.1.224:1880/ui/#!/1 → navigate to page 1
On touch screens, you may also use “wiping” for changing the page.
Node-Red is running over MQTT. This service first has to be installed (e.g. Mosquitto).

The following code writes to Node-Red topics (marked green):

```
151             await publish(self.js, "emv.garage.data.input", json.dumps(self.dataInput
152             mqttlib.publish(self.client_mqtt, "emv/garage/data/input", json.dumps(sel
153
154             await publish(self.js, "emv.garage.data.wallbee", json.dumps(dataEV).enco
155             mqttlib.publish(self.client_mqtt, "emv/garage/data/wallbee", json.dumps(d
```

Above the code may listen to the corresponding topics in runtime. For that callbacks are defined (marked orange):

```
111         await self.client.subscribe('emv.garage.wallbee.auto', cb=on_message_auto)
112         await self.client.subscribe('emv.garage.wallbee.setpoint', cb=on_message_setpoint)
113         self.client_mqtt.loop_start()
```

The callbacks may run some code when called (example with async io):

```
87      async def runnIO(self):
88
89          async def on_message_auto(msg):
90              print(f"New Automatic Data '{msg.subject} {msg.reply}: {msg.data.decode()}")
91              if msg.data.decode() == '"false"':
92                  self.modeman.set_manual(True)
93                  print('Bedienung manuell')
94              if msg.data.decode() == '"true"':
95                  print('Automatik')
96                  self.modeman.set_manual(False)
97                  self.modeman.set_on(False)
98
99          async def on_message_setpoint(msg):
100             print(f"New setpoint Data '{msg.subject} {msg.reply}': {msg.data.decode()}")
101             if self.modeman.get_manual():
102                 dataEINT = int(msg.data.decode())
103                 self.modeman.set_setpoint(dataEINT)
```

6 External Hardware for Installation in Buildings

TODO: Replace this chapter with Link to SGr test lab documentation.

Maybe a part of the test lab documentation according the hardware setup should be delivered within OpenCEM documentation.

Device / Type	Manufacturer	Interface	Prio
3phase Energy Meter ABB B23 112-100	ABB	Modbus® RTU	1
3phase Energy Meter ABB B23 312-100 bidirectional	ABB	Modbus® RTU	1
3phase Energy Meter ABB B23 212-100 bidirectional	ABB	Modbus® RTU	2
Relays FSR14-2x (over FAM14)	Eltako	EnOcean®	3
3phase Energy Meter DSZ14	Eltako	EnOcean®	
Room Temperature Sensors FTF65	Eltako	EnOcean®	3
Button 1way F1FT65	Eltako	EnOcean®	3
Button 2way F4FT65	Eltako	EnOcean®	3
Room Temperature Sensors WRF04	Thermokon	Modbus® RTU	1.5
Storage Tank Temperature Sensors SR65 TF/AKF (cable/fixed sensor)	Thermokon	EnOcean®	3
3phase Energy Meter 3EM	Shelly	REST API	1.6
Smart Relay 2PM / 4PM	Shelly	REST API	1.6
Room Temperature and Humidity Sensor H&T	Shelly	REST API	1.7
Thermostatic Radiator Valve TRV	Shelly	REST API	1.7
Button 1way button1	Shelly	REST API	1.7
SmartPlug PLUGS	Shelly	REST API	3
SmartBulp DUO RGBW	Shelly	REST API	3
Digital IOs / Relays	WAGO	Modbus TCP	4
Temperature Sensor Inputs	WAGO	Modbus TCP	4
Eco EV Charging Station Type 2 (AC 16A) 4..11 kW (3phase), 1.4..3.7 kW (1phase)	Wallbe Compleo	Modbus TCP	1.8
Pro EV Charging Station Type 2 (AC 32A) 4..22 kW (3phase)	Wallbe Weidmüller	Modbus TCP	1.8
Heat Pump Brine-to-Water Inverta Eco	CTA	Modbus TCP	2
Heat Pump Brine-to-Water Inverta TWW	CTA	Modbus TCP	2
Heat Pump Air-to-Water WPL15/25 (AC)	Stiebel Eltron	Modbus TCP	2
Heat Pump ???	Hoval	Modbus TCP	3
Heat Pump ???	Alpha Innotec	Modbus TCP	3
Battery Powerwall 1	Tesla SolarEdge	Modbus RTU Modbus TCP?	4
PV Inverters ???	SolarEdge SMA, etc.	Modbus TCP SunSpec®	4

6.1 Implementation of sensors/actuators (Sergio)

In this chapter, the devices that are already implemented in OpenCEM are discussed.

6.1.1 ABB electricity meter

As of this point, the ABB B23 112-100 and ABB B23 312-100 have been implemented in OpenCEM. The two meters differ in that the ABB B23 312-100 can measure in both directions while the other model can only measure in one direction. The bidirectional meter also has energy registers that can be reset. The meters can measure up to 3 phases. They also have a user interface with a display, which allows for various settings to be made. The communication between the meters and the OpenCEM library is done via the Modbus RTU protocol. Additionally, the meters are also capable of communicating over M-Bus. The meters will be connected to the system via a RS485 to USB adapter.

For the communication over Modbus RTU the following configurations must be made. The system has been tested with these settings:

- Baudrate: 19200
- Parity: Even
- Address: 1 (for ABB B23 112-100)
- Address: 2 (for ABB B23 312-100)

This illustration shows how to configurate the ABB electricity meters:

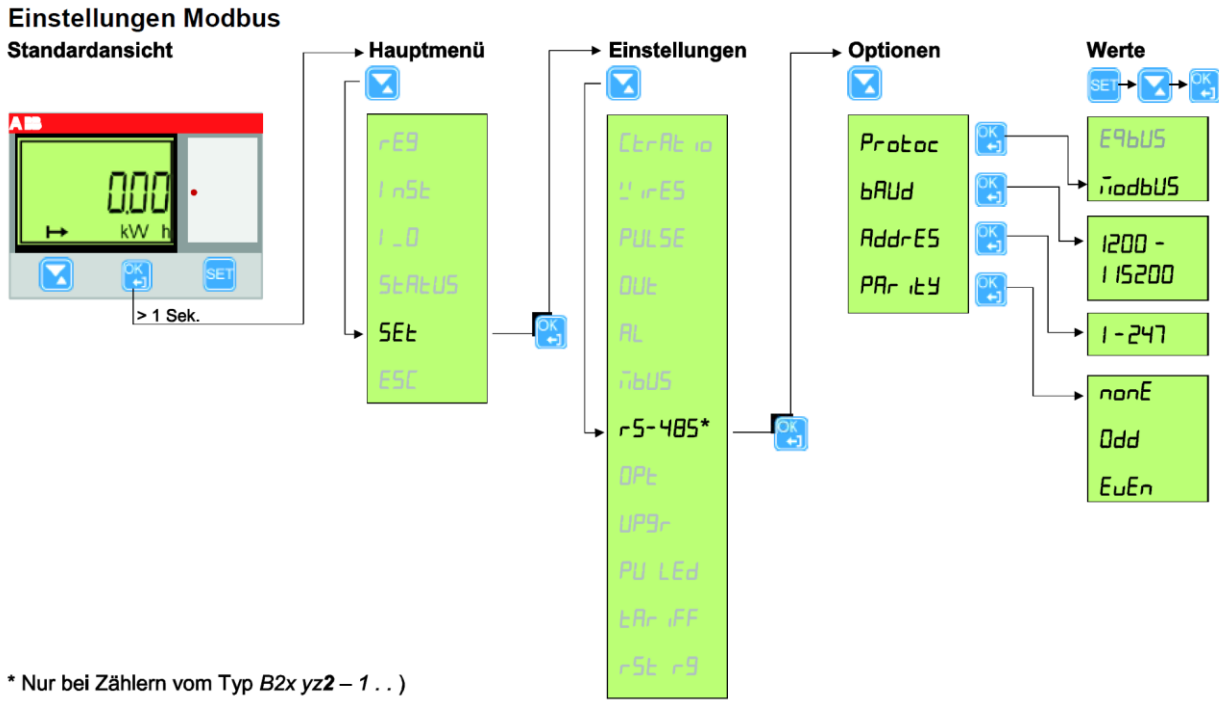


Figure 1 Performing Modbus configurations on the ABB power meter (Reference: Doku P5 S.F)

How to initialize the ABB electricity meters in OpenCEM?

First, the hardware port that the RS485 adapter is connected to must be identified. This information can be found in the device manager. It will be for example COM7. For the wiring of the adapter see chapter 6.3.

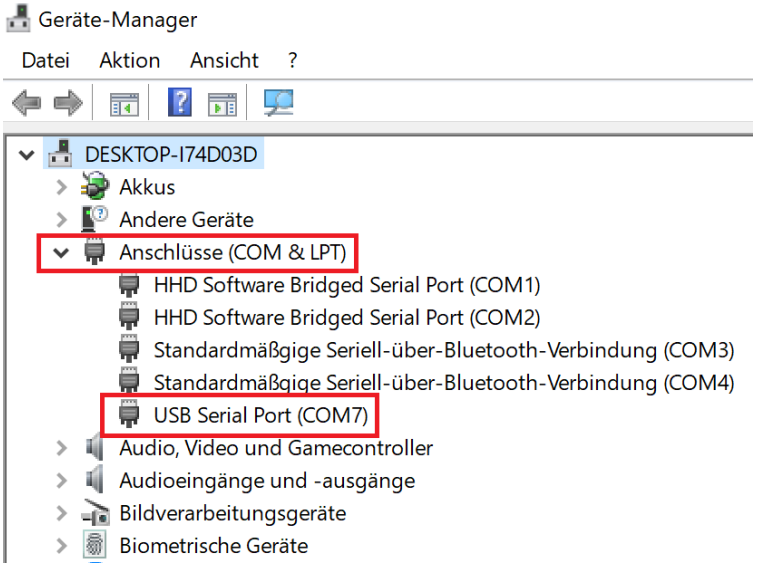


Figure 2 Determining the port for Modbus RTU communication in the device manager (Reference: Doku P5 S.F.)

Initialization via SmartGridready XML-file

For the initialization of the ABB electric meters, the following XML files are available, which can be loaded in OpenCEM.

- SGr_04_0016_xxxx_ABBMeterV0.2.1_Uni.xml
- SGr_04_0016_xxxx_ABBMeterV0.2.1_Bi.xml

```
from OpenCEM.cem_lib_components import power_sensor
# Initialize the sensor using SGr-XML file
AbbUni = power_sensor(is_smartgridready=True, id=1,
    has_energy_import=True,
    has_energy_export=False,
    XML_file="xml_files/SGr_04_0016_xxxx_ABBMeterV0.2.1_Uni.xml")
```

Initialization via name

For the ABB B23 112-100 and ABB B23 312-100, the device information is stored in the OpenCEM library and they can be initialized by their name. It is important to note that the SGr devices must be initialized before the native devices. This must be done because only one RTU client can exist on the same hardware port. This means that for all RTU devices that communicate via the same port, only one instance of an RTU client can be created. If all devices are initialized via XML file, this is automatically checked. When using native initialization, the global RTU client must be passed like shown in the following code snippet.

```
from OpenCEM.cem_lib_components import power_sensor, OpenCEM_RTU_client
# determine global RTU client
RTU_global_client = OpenCEM_RTU_client(
global_client = AbbUni.get_pymodbus_client()).get_OpenCEM_global_RTU_client()

#  Initialize ABB B23 312-100 via name
AbbBi = power_sensor(is_smartgridready=False, id=2, has_energy_import=True,
has_energy_export=True, name="ABB B23 312-100",
manufacturer="ABB", bus_type="RTU", client=RTU_global_client)
```

Initialization manually

For electric meters that are not implemented, a manual method has also been developed for communicating with them via Modbus RTU. To do this, information about the Modbus registers must be provided and initialized like in the following code snippet.

```
from OpenCEM.cem_lib_components import power_sensor, OpenCEM_RTU_client

# Create global RTU client (only if no SmartGridready device is available)
RTU_global_client = OpenCEM_RTU_client().get_OpenCEM_global_RTU_client()

myPowerSensor = power_sensor(is_smartgridready=False, id=2,
has_energy_import=True,has_energy_export=True, name="Stromzähler",
manufacturer="ABB", bus_type="RTU", client=RTU_global_client)

# Add Modbus register manually (start address, length, unit, data type,
scaling factor)
myPowerSensor.add_RTU_Power_entry(23316, 2, "WATTS", "int32", 0.01)
myPowerSensor.add_RTU_EnergyImport_entry(20480, 4, "KILOWATT_HOURS",
"int64_u", 0.01)
myPowerSensor.add_RTU_EnergyExport_entry(20484, 4, "KILOWATT_HOURS",
"int64_u", 0.01)
```

Read data from electricity meter

The following methods are used to read data from all power sensors in OpenCEM, regardless of the type of initialization or bus type used (exception 3EM over Shelly Cloud).

```
from OpenCEM.cem_lib_components import power_sensor

# Initialize the ABB electricity meter with XML
AbbUni = power_sensor(is_smartgridready=True, id=2, has_energy_import=True,
has_energy_export=True,
XML_file="xml_files/SGr_04_0016_xxxx_ABBMeterV0.2.1_Bi.xml")

# readout of instantaneous power, energy import and energy export
print(f"Power: {AbbUni.read_power()}")
print(f"Import: {AbbUni.read_energy_import()}")
print(f"Export: {AbbUni.read_energy_export()}")
>>>Output:
Power: (0.0083, 'KILOWATT', 0)
Import: (0.61, 'KILOWATT_HOURS', 0)
Export: (0.0, 'KILOWATT_HOURS', 0)
```

6.1.2 Shelly Devices

Shelly is a company that offers intelligent home automation products. Various sensors or actuators are offered by Shelly. The available sensors include, for example, power meters, temperature and humidity sensors, as well as motion sensors. These devices are able to provide the measured data in real-time. The Shelly devices implemented in OpenCEM are presented in this chapter.

In order for Shelly devices to make use of the OpenCEM library, some information about the device is first necessary. In addition, a connection from the device to the Wi-Fi (or LAN) must be established. This can be done with the Shelly Smartphone App. (https://www.shelly.cloud/de-ch/shelly-app)

How to add devices to the Shelly App?

The following illustration shows how to add devices to the Shelly App and Cloud.

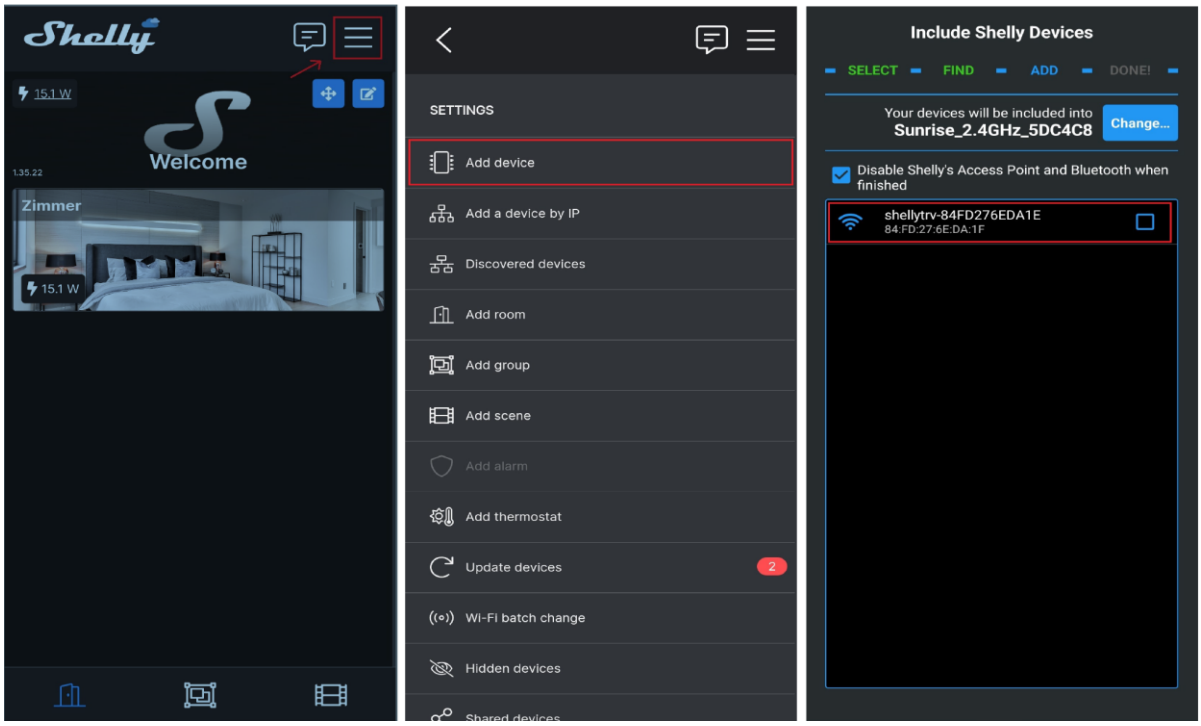


Figure 3 Adding Shelly devices in the Shelly app. Step 1: Open the menu, Step 2: Select 'Add device', Step 3: Available devices will be displayed. Select the device and follow the app's instructions (Reference Doku P5 S.F.).

After the device is added the *Device IP* (local communication) or the *Device ID* (Shelly Cloud) must be determined to use the devices in OpenCEM. This can be done like shown in the next picture.

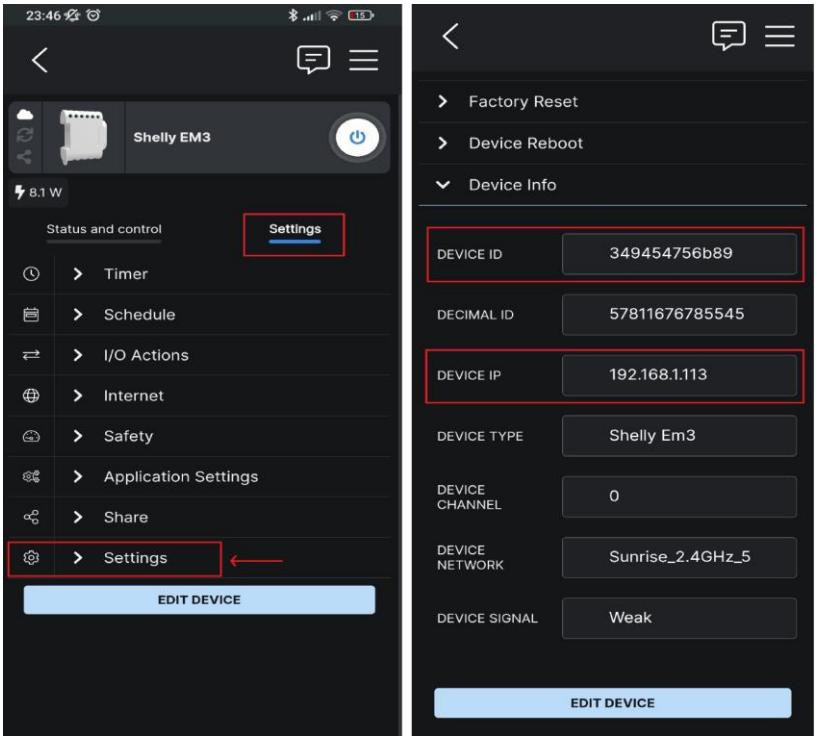


Figure 4 Retrieving the Device IP and Device ID from the Shelly App. Step 1: Open the Settings menu, Step 2: Select Device Info (Reference: Doku P5 S.F)

Communication over Shelly cloud

The Shelly cloud allows products to be accessed worldwide via the internet. This functionality is particularly exciting for larger buildings or, for example, buildings with different apartments. These buildings may have multiple Wi-Fi networks. Therefore, communication via the cloud could be a way to connect all Shelly devices in these buildings. Communication via the cloud requires knowing the *server address* of the cloud and the *device IDs* of the desired devices. In addition, the *authorization cloud key* must be known. This information can be obtained via the Shelly app and is shown in the following picture.

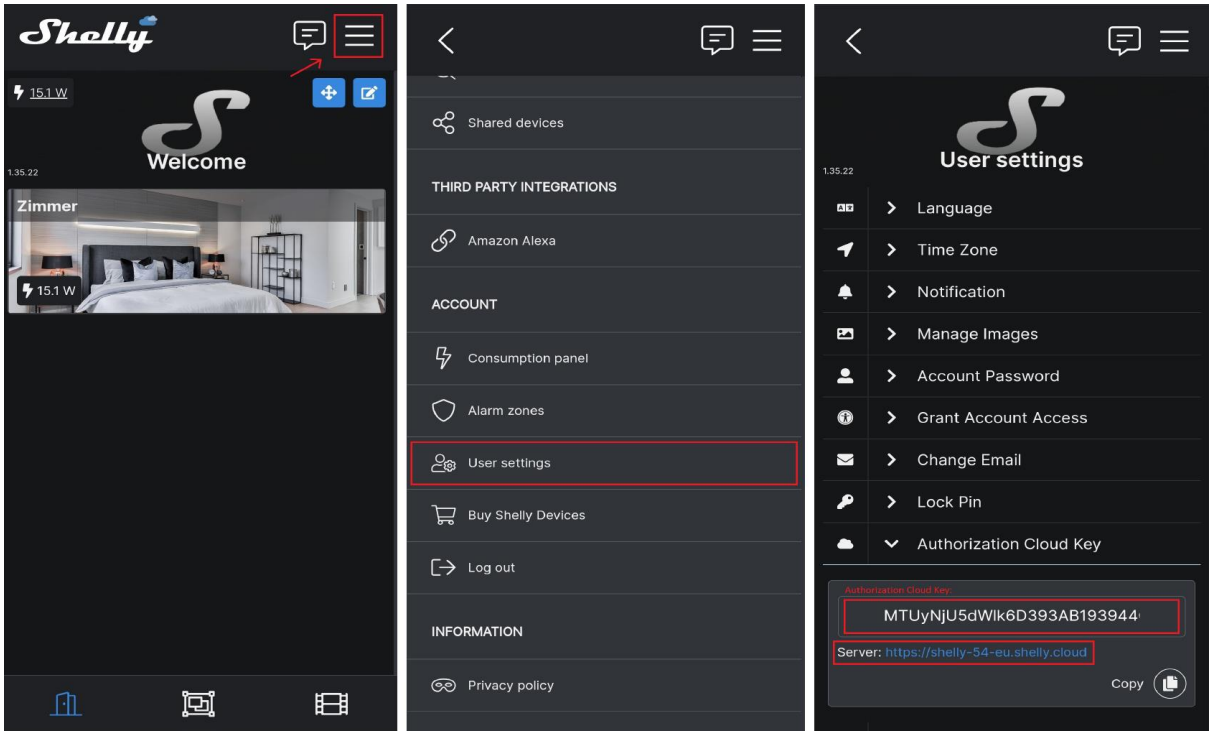


Figure 5 Reading the Cloud server and Key from the Shelly App. Step 1: Open the menu, Step 2: Open User Settings, Step 3: Key and Server visible under Authorization Cloud Key (Reference: Doku P5 S.F)

The server address and authorization cloud key must be stored in the file cem_lib_components.py, like shown in this code snippet.

```
"""IMPORTS"""

# Authentication Key and corresponding server for the Shelly Cloud
shelly_auth_key = "MTUyNjU5dWlk6D393AB193944CE2B1D84E0B573EAB..."
shelly_server_address = "https://shelly-54-eu.shelly.cloud/"
```

Important! Methods that use the Shelly cloud can only be used once a second. This limitation is set by Shelly for their cloud. There are special methods that can be used while using the cloud that minimize the request made. For example *read_all_channels() for reading all the channel values of a relay, instead of read_channel(channel_nr).*

6.1.3 Shelly 3EM

The Shelly 3EM is a bidirectional indirect power meter from the manufacturer Shelly. It has 3 channels (max. 120 Amperes per channel). This makes it possible to measure a 3-phase system or 3 independent circuits. The communication is done via Wi-Fi. This device has not only the measurement function but also the ability to switch the integrated relay (max. 10 Ampere). The implemented methods measure the total values for the electricity meter. For the local communication the following methods are available:

```
from OpenCEM.cem_lib_components import shelly_power_sensor

# Initialization Shelly 3EM via local address (default)
myPowerSensor = shelly_power_sensor("192.168.1.113")

# readout of instantaneous power, energy import and energy export
print(f"Power: {myPowerSensor.read_power()}")
print(f"Import: {myPowerSensor.read_energy_import()}")
print(f"Export: {myPowerSensor.read_energy_export()}")
>>>Output:
Power: (0.0087, 'KILOWATT', 0) # value, unit, error code
Import: (0.3798, 'KILOWATT_HOURS', 0)
Export: (0.0003, 'KILOWATT_HOURS', 0)
```

For the communication over the Shelly Cloud:

```
from OpenCEM.cem_lib_components import shelly_power_sensor

myPowerSensorCloud = shelly_power_sensor("349454756b89",
    bus_type="SHELLY_CLOUD") # use Device ID

# readout of instantaneous power, energy import and energy export
(power, energy_import, energy_export) = myPowerSensorCloud.read_all()
print(f"Power: {power}")
print(f"Import: {energy_import}")
print(f"Export: {energy_export}")
>>>Output:
Power: (0.0088, 'KILOWATT', 0)
Import: (0.3784, 'KILOWATT_HOURS', 0)
Export: (0.0003, 'KILOWATT_HOURS', 0)
```

6.1.4 Shelly Relays (Pro 2PM and Pro 4PM)

Shelly offers a variety of different relays. For OpenCEM, the Pro 2PM and Pro 4PM were implemented and tested. However, the methods created should be able to be used for all Shelly relays.

Device Info: Shelly Pro 2PM Channels: 2 (independent) Communication: LAN, WLAN Max. 16 A per channel, or total max. 25 A Can measure power	Device Info: Shelly Pro 4PM Channels: 4 (independent) Communication: LAN, WLAN, Bluetooth Max. 16 A per channel, or total max. 25 A Can measure power Physical GUI
	
Figure 6 Shelly Pro 2PM (Reference: https://www.shelly.cloud/de-ch/products/)	Figure 7 Shelly Pro 4PM (Reference: https://www.shelly.cloud/de-ch/products/)

Use the Shelly Relays over the local communication:

```
from OpenCEM.cem_lib_components import shelly_relais_actuator
# Initializing Shelly 2PM via local address (default)
myRelais = shelly_relais_actuator("192.168.1.112", 2) # IP, Number of
channels

# Turn on channel
myRelais.write_channel(1, "on") # Channel number starts at 0

# Turn off channel
myRelais.write_channel(1, "off")

# Read status of a single channel
print(myRelais.read_channel(1))
>>>Output: False

# Read status of all channels
myRelais.read_all_channels()
>>>Output: [True, False]
```

Following code snippet shows how to use the shelly relays over the Shelly Cloud. Individual channels cannot be read. They are always read together with read_all_channels().

```
from OpenCEM.cem_lib_components import shelly_relais_actuator
# Initializing Shelly 4PM via Shelly Cloud
myRelais_cloud = shelly_relais_actuator("30c6f7837220", 4,
bus_type="SHELLY_CLOUD") # Device ID, Number of channels

# Turn on channel
myRelais_cloud.write_channel(0, "on") # Channel number starts at 0

# Turn off channel
myRelais_cloud.write_channel(0, "off")

# Read status of all channels, reading a single channel is not possible via
cloud
myRelais_cloud.read_all_channels()
>>>Output: [False, True, False, False]
```

6.1.5 Shelly H&T

The Shelly H&T is a battery-operated temperature and humidity sensor that can also be powered by a 3.3 volt power source or the optional USB power supply. As this sensor is battery-powered, it does not keep a local server open for communication all the time. With a press on the button in the inside of the sensor, a local server gets turned on for the device and adjustments to the settings can be made.

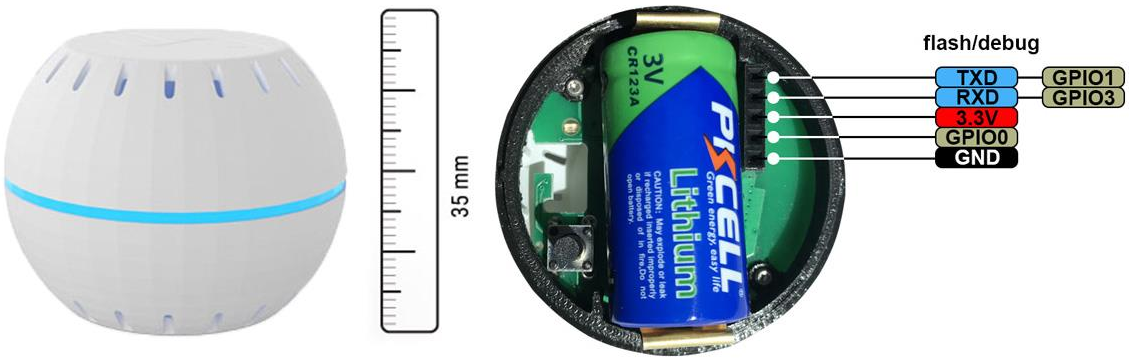


Figure 8 Shelly H&T with pin connections (Reference: https://www.bastelgarage.ch/shelly-ht-weiss-humidity-temperatur-sensor)

The sensor sends a notification to the Shelly cloud when its values have changed. The temperature threshold is important. This parameter indicates at what temperature change a new value

is written to the cloud. The following code snipped shows the use of the H&T Sensor in OpenCEM:

```
from OpenCEM.cem_lib_components import shelly_temp_and_hum_sensor
# Initializing Shelly Temperature and Humidity sensor with Shelly Device ID
mySensor = shelly_temp_and_hum_sensor("701f93")

# Reading and printing temperature
temp, timeStamp, valid = mySensor.read_temperature()
print(f"Temperature is: {temp} C°")
>>>Output: Temperature is: 13.88 C°
```

6.1.6 Shelly Button 1

The Shelly Button 1 is a battery-powered Wi-Fi button from Shelly. It can also be powered by a micro-USB power supply. It can be used to control various smart home devices and up to 4 different actions can be programmed on it (single, double, triple press and long press). When pressed, multiple http requests can be sent from the button and thus something can be controlled. For OpenCEM a demo server has been programmed that receives requests from the button (Shelly_Button_Server.py).

Figure 9 Shelly Button 1

The following illustration shows how the HTTP requests are set in the button's settings.

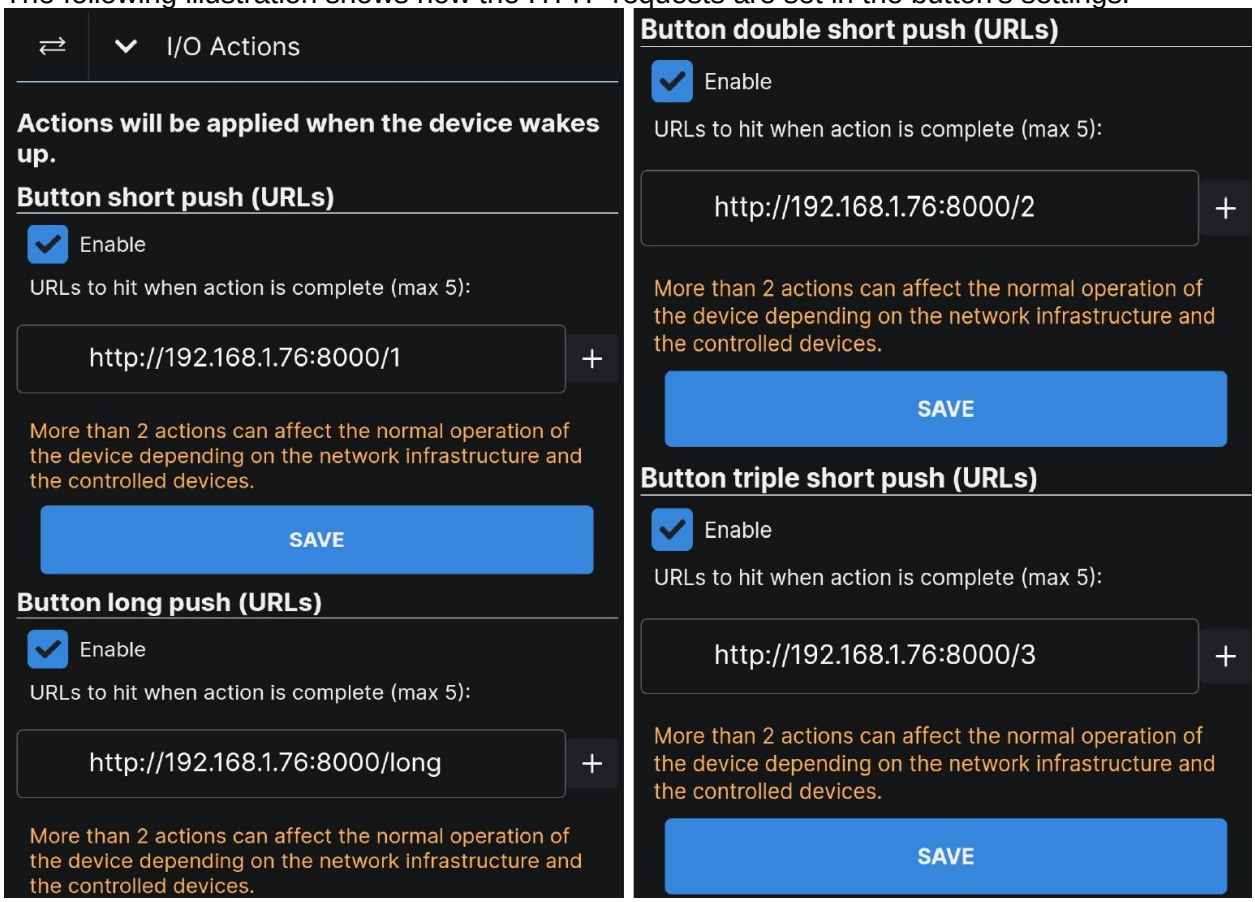


Figure 10 Configuring the Shelly Button in the Shelly App (Reference: Doku P5 S.F.)

6.1.7 Electric Meter Modbus (unidirectional)

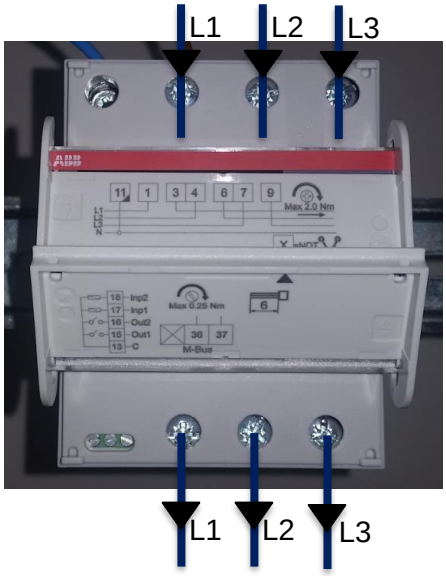
For the measurement of single producers or consumers unidirectional meters are used.

ABB electric meter 3 phase, MID certified, type B23 112-100 (unidirectional).

The following figures show the wiring for a producer (e.g. pv plant) or consumer (e.g. heat pump).

Connection of the three phases L1, L2, L3 for measuring a producer (pv plant)

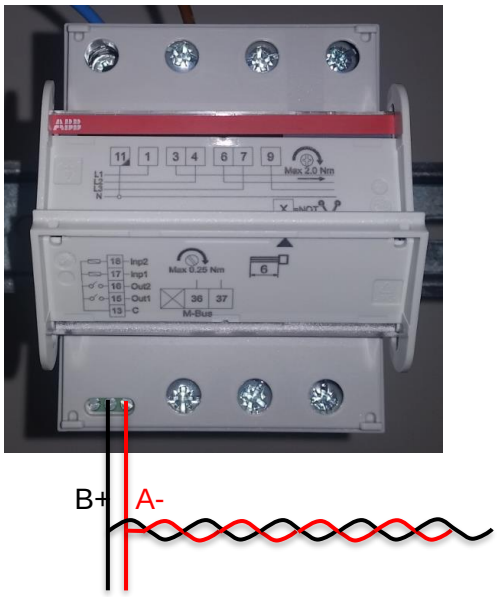
from the central distribution



to a single consumer

Connection of the three phases L1, L2, L3 for measuring a consumer

The meters are connected to each other in 2-wire technology via Modbus RTU.



Modbus wiring (2-wire A/B)

6.2 Electric Meter Modbus (bidirectional)

For bidirectional measurement of the net consumption at the main connection, a bidirectional meter is used. The bidirectional meter is installed in series with the bidirectional electricity meter of the utility company (EVU).

ABB electric meter 3 phase, MID certified, type B23 312-100 (bidirectional).

The connections are located at the top and bottom behind the covers. The current flow runs in two ways. The figure below shows the connection as main bidirectional meter.

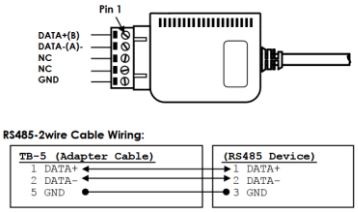
Connection of the three phases L1, L2, L3 for bidirectional measurement

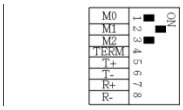
6.3 Modbus Wiring and Modbus Gateway

A line topology (serial wiring) is used for the cabling of Modbus RTU / RS485. You should always use a shielded cable to avoid EMC interference. The cables should not be laid together with current-carrying lines. In standard RS485 a maximum of 32 participants are allowed on one bus.

Modbus RTU is connected via an RS485-USB converter:

RS485 USB converter





DIP switch settings (source: ExSys)

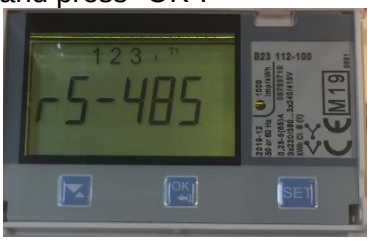
Connection via USB converter

Connection via RJ45 DIN-mounting clamp

6.3.1 Configuration of the power meters

The meters in the bus must be assigned unique addresses (IDs). For ABB series 23 meters the bus address is set via the display as follows:

1. select "SET" in the main menu and press "OK".



2. select "RS-485" with the arrow keys on the left and press "OK".
3. Press the left arrow key 1x and go to the menu "bAUd". The display will show the set baud rate. Set the baud rate to 19’200 (or the desired value).
4. Press the left arrow key 1x and go to the next menu "AddrES".

The display will show the current unit address. Enter a unique unit address here. Press the "Set" key to start the configuration. Press "OK" to go to the next digit. If all 3 digits are set, the display jumps to the main view after a few minutes.
5. Press the left arrow key 1x and go to the next menu "Parity". Set the "Parity" to Even (or the desired value).

6.3.2 Connection of electric vehicle charging stations

The following charging stations are currently supported:

Charging Stations by Wallbe (left Pro, right Eco)

Charging Stations by Weidmüller (left), Compleo (right)

The charging stations contain the controller from wallbe. The variants from Weidmüller and Com-pleo are "labelled" variants, the OEM is wallbe.

The electric vehicle charging stations communicate via Modbus TCP.

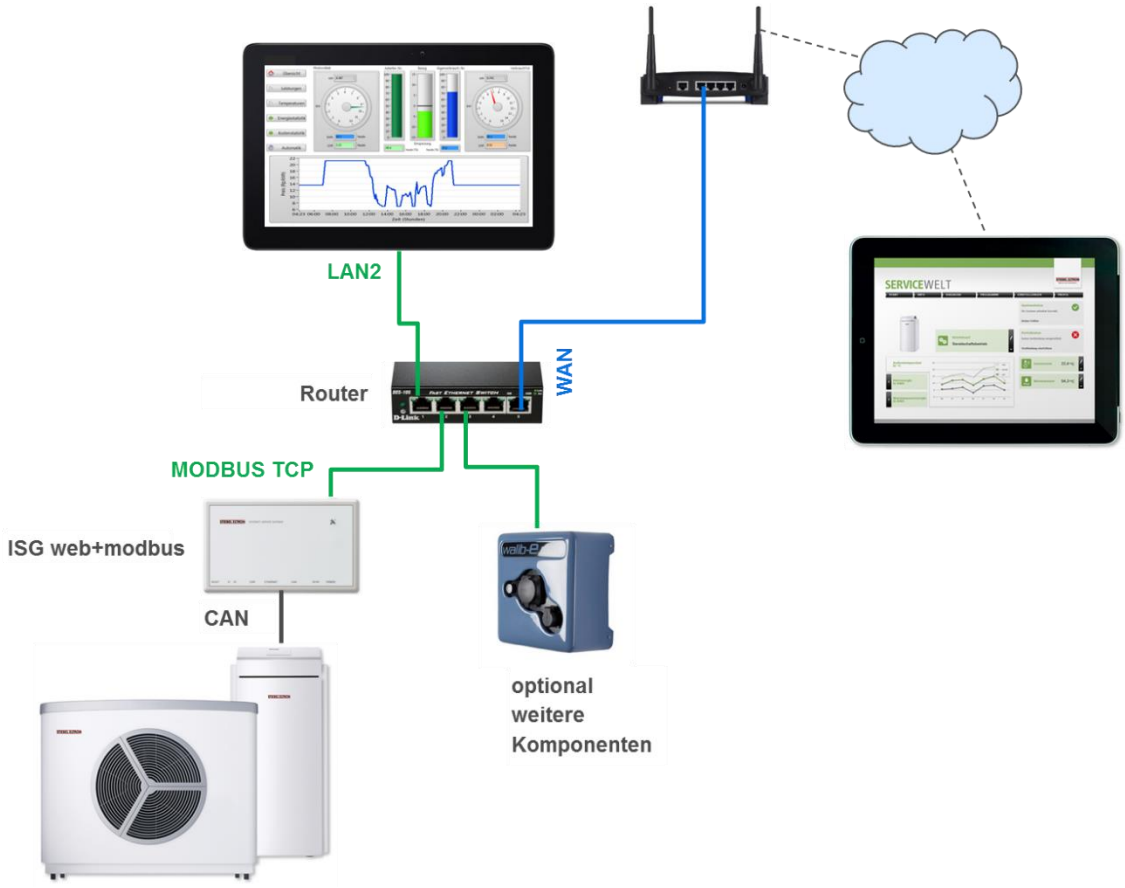
List of supported charging stations and power levels:

Stromstärke (Absicherung)	**1-phasig**	**3-phasig**	**Bemerkung**
32A	7.4 kW	22 kW Wallbe Pro Weidmüller	MFH, Areale
16A	3.7 kW Wallbe Eco Compleo	11 kW Wallbe Eco Compleo	EFH
6A	1.4 kW	4 kW	Minimale Ladeleistung (regelbar)

6.3.3 Connecting Heat Pumps via Modbus

6.4 Stiebel Eltron with ISG extension

The heat pump communicates via Modbus TCP. Stiebel Eltron requires the ISG (Internet Service Gateway) accessory in the "web+modbus" variant. The ISG can be integrated directly into the home network. More robust is the integration via a separate router, as shown in the picture below. In this case, the router must be connected to the home network via its WAN connection.

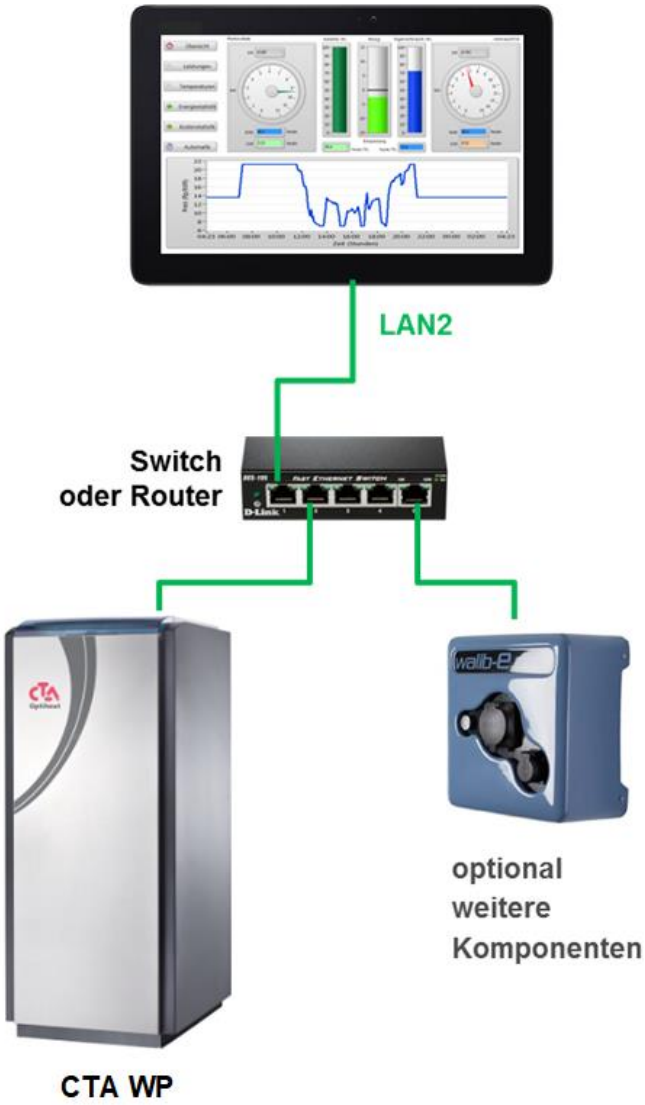


Connection of Heat Pump Stiebel Eltron

The heat pump must be connected to the internet so that it can be accessed via the Stie-bel Eltron service world (cloud). The ISG has an integrated local web server and is configured via the browser. This also includes setting the IP address.

6.5 CTA Inverta or TWW over Modbus TCP

The heat pump communicates via Modbus TCP. The heat pump controller must have software version 1.6.4 or higher. The heat pump is connected to energy manager via LAN cable and optionally via the switch/router.



Connection of heat pump CTA.

6.5.1 Wired Room Temperature Sensors (Modbus RTU)

For measuring the room temperature in larger buildings, wired sensors from Thermokon are used (Abb. 1).

6.5.1.1 Thermokon WRF04 (old)

Abb. 1 Thermokon room sensor WRF04 for surface mounting (without/with display)

The sensors are connected to 2-wire Modbus RTU (RS485) for data transmission and require a 24Vdc or 24Vac power supply (Abb. 2). WRF04 is not delivered anymore by Thermokon.

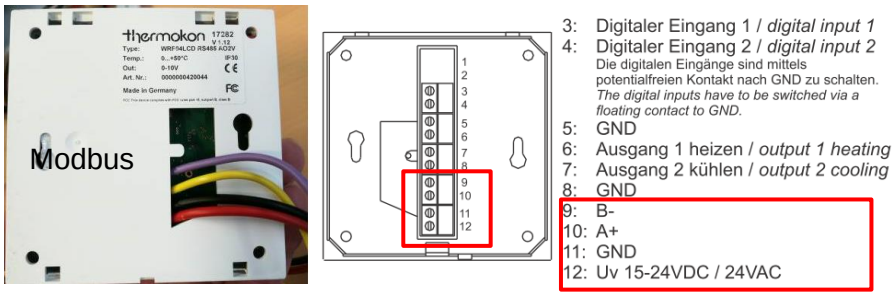


Abb. 2 Spannungsversorgung und Modbus-Anschluss (Quelle: Thermokon)

6.5.1.2 Thermokon NOVOS 3 (new)

Abb. 3 Thermokon room temperature sensor Novos 3 (without display)

NOVOS is the official new product line which replaces the WRF… family. The Modbus register addresses and wiring are equal (according to the support of Thermokon)

6.6 Logging in OpenCEM

For OpenCEM, three different loggers are available, which can be used for different purposes.

Initialization Method	Logging Level	File Name	Function
create_event_logger()	INFO (20)	Event.log	Logs system events such as warnings or informations.
create_statistics_logger()	Custom	Statistics.log	Logs values and states of sensors/actuators.
create_debug_logger()	DEBUG (10)	Debug.log	Logs everything and is intended to be useful for a potential debugging process.

The methods to initialize the loggers should be called at the beginning of the code. The logs are dated at the end of the day and saved as text files in the _logFiles folder.

How to add entries to the Loggers?

Event logger: with logging.info(“message), logging.warning(“message”), etc.

Statistics logger: entries get added automatically when reading from a device. To disable this put *is_logging* = *False* as a parameter, when initializing a device in OpenCEM.

Debug logger: with logging.debug(“message”)

Display logs in console

The log entries are not displayed in the system console by default. To display them, the function show_logger_in_console(Log_level) must be called after initializing the loggers. The function argument Log_level specifies which logs will be displayed in the console.

- 10 = Debug
- 20 = Info
- 30 = Warning
- 40 = Error
- 50 = Critical

Import the statistics logs in Excel

The statistics log is saved as a text file and can be imported into Excel as show. The semicolon is used as a delimiter. The headers are saved in the text file and are displayed correctly in Excel.

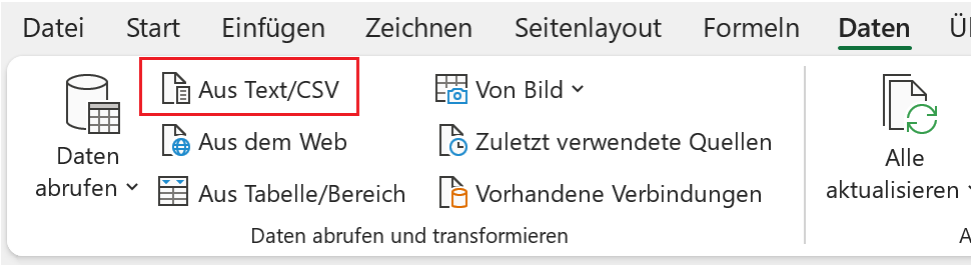


Figure 11 Import statistics log in Excel (Reference: Doku P5 S.F.)

An imported statistics log is represented in Excel as follows:

	A	B	C	D	E	F	G	H	I	J
1	timestamp	actuator/sensor-type	name	bus-type	is_smartgridready	id	value_name/channel	value	unit	last_updated
2	2022-12-27 10:47:35,369	power	SmartMeter B23	RTU	WAHR	1	POWER	0.0078	KILOWATT	
3	2022-12-27 10:47:35,621	power	ABB B23 312-100	RTU	FALSCH	2	POWER	0.0089	KILOWATT	
4	2022-12-27 10:47:35,731	power	Shelly 3EM Powermeter	SHELLY_LOCAL	FALSCH	192.168.1.113	POWER	0.0083	KILOWATT	
5	2022-12-27 10:47:35,986	power	SmartMeter B23	RTU	WAHR	1	ENERGY_IMPORT	0.27	KILOWATT_HOURS	
6	2022-12-27 10:47:36,253	power	ABB B23 312-100	RTU	FALSCH	2	ENERGY_IMPORT	0.39	KILOWATT_HOURS	
7	2022-12-27 10:47:36,523	power	ABB B23 312-100	RTU	FALSCH	2	ENERGY_EXPORT	0.0	KILOWATT_HOURS	
8	2022-12-27 10:47:36,659	power	Shelly 3EM Powermeter	SHELLY_LOCAL	FALSCH	192.168.1.113	ENERGY_IMPORT	0.2751	KILOWATT_HOURS	
9	2022-12-27 10:47:36,701	power	Shelly 3EM Powermeter	SHELLY_LOCAL	FALSCH	192.168.1.113	ENERGY_EXPORT	0.0003	KILOWATT_HOURS	
10	2022-12-27 10:47:36,850	temperature	Shelly T&H	SHELLY_CLOUD	FALSCH	701f93	TEMPERATURE	15.38	CELSIUS	27.12.2022 02:14
11	2022-12-27 10:47:36,937	relais	2PM	SHELLY_LOCAL	FALSCH	192.168.1.112	1	False		
12	2022-12-27 10:47:36,981	relais	2PM	SHELLY_LOCAL	FALSCH	192.168.1.112	1	True		
13	2022-12-27 10:47:37,068	relais	4PM	SHELLY_LOCAL	FALSCH	192.168.1.114	0	False		
14	2022-12-27 10:47:37,094	relais	2PM	SHELLY_LOCAL	FALSCH	192.168.1.112	1	True		
15	2022-12-27 10:47:37,116	relais	4PM	SHELLY_LOCAL	FALSCH	192.168.1.114	0	False		
16	2022-12-27 10:47:47,401	power	SmartMeter B23	RTU	WAHR	1	POWER	0.0086	KILOWATT	
17	2022-12-27 10:47:47,657	power	ABB B23 312-100	RTU	FALSCH	2	POWER	0.0	KILOWATT	
18	2022-12-27 10:47:47,710	power	Shelly 3EM Powermeter	SHELLY_LOCAL	FALSCH	192.168.1.113	POWER	0.0017	KILOWATT	

6.7 Experimental setup

To test the functions of the devices used in this project, an experimental setup was created. The setup shown in the next picture is the final setup of P5 (as of January 1, 2023). There are two power cables at the output terminals that can be swapped between terminals as desired. When creating the setup, it was ensured that enough space for future expansions is available. With this test setup and the Python script CommunicationTest_all_devices.py, the implemented functions of the OpenCEM library can be tested.

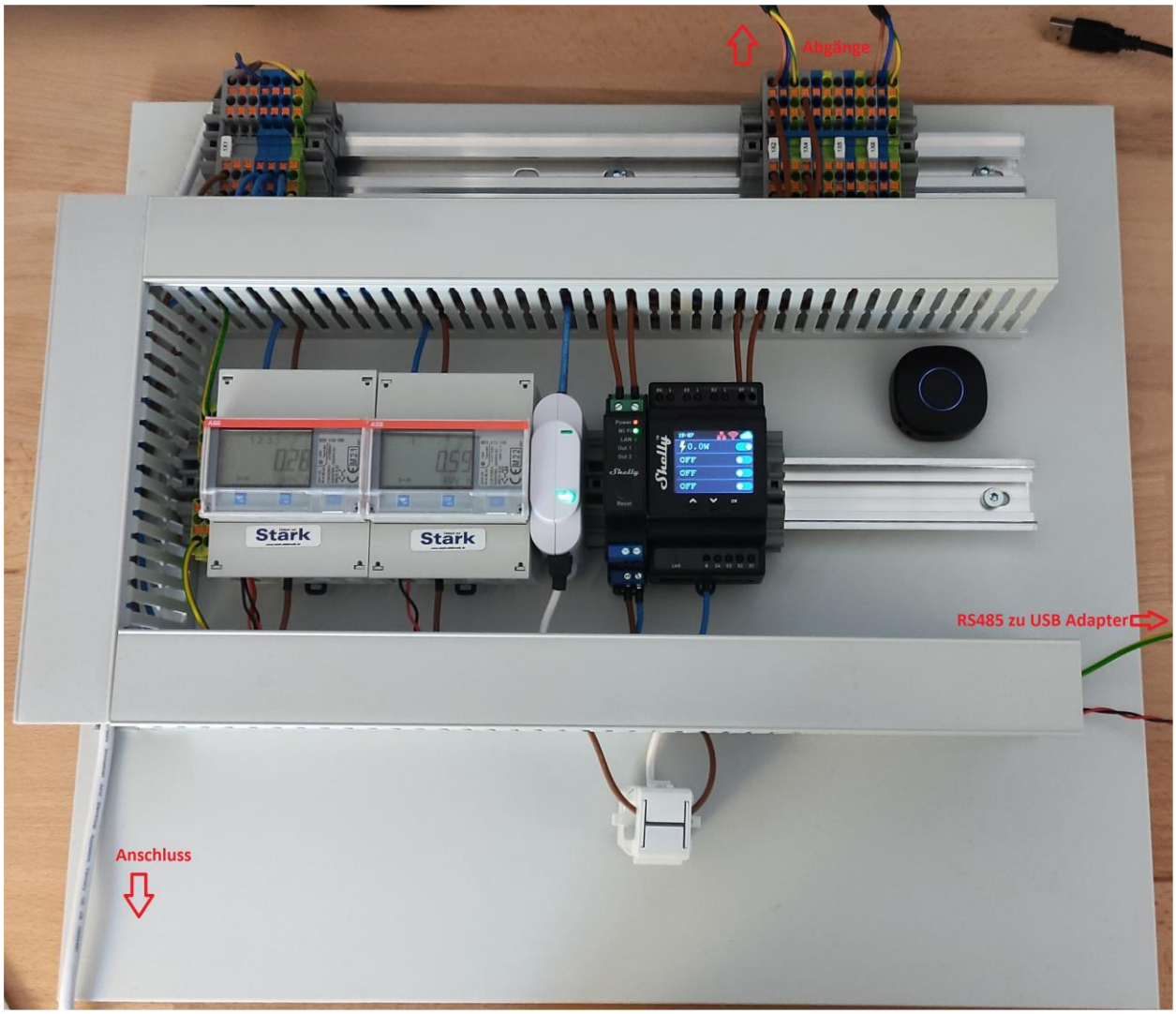


Figure 12 Final experimental setup of the P5 (Reference: Doku P5 S.F)

The following illustration shows the wiring diagram for the experimental setup.

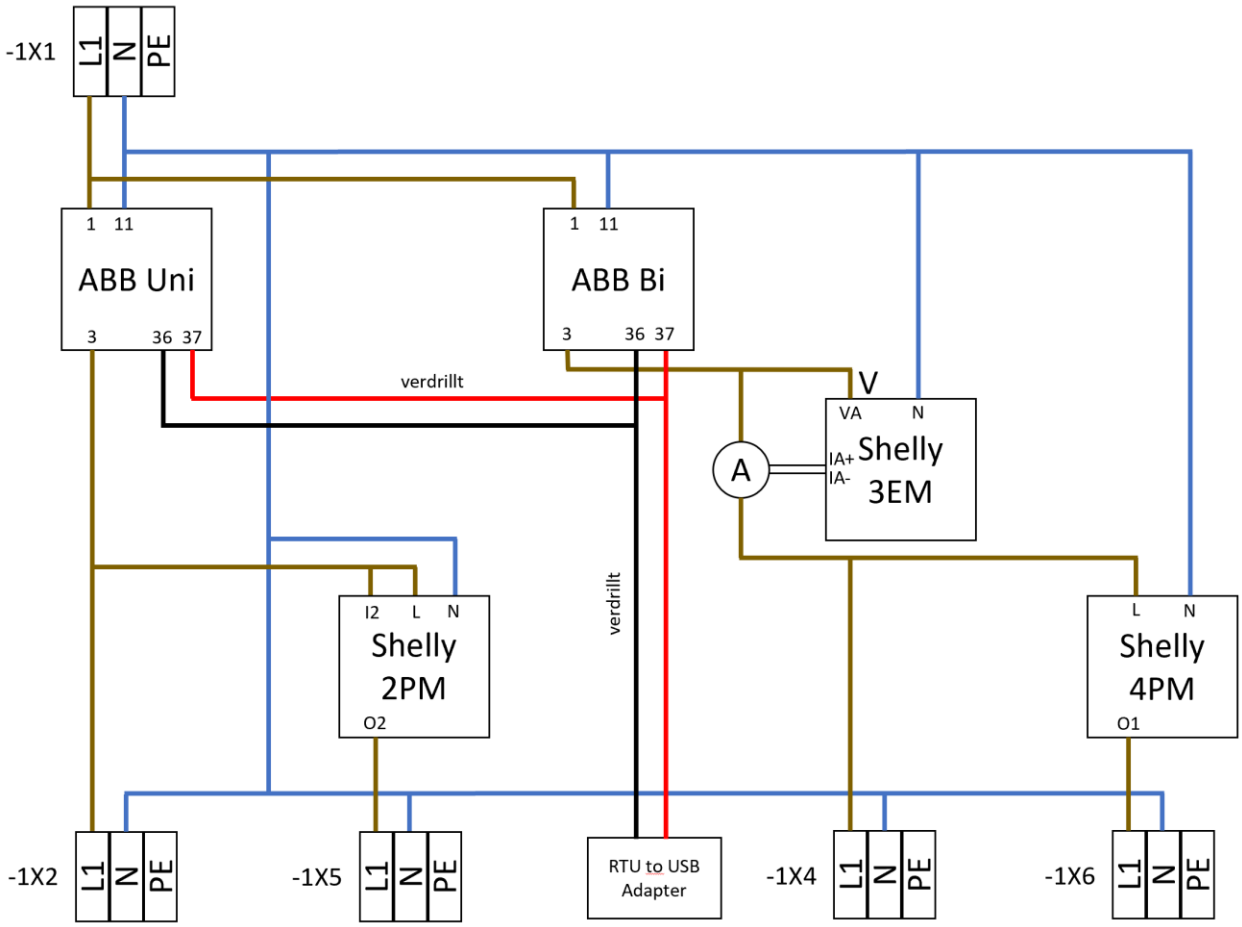


Figure 13 Wiring diagram of the experimental setup (Reference: Doku P5 S.F)

6.7.1 The wiring of the individual devices

6.7.1.1 Shelly 3EM

The Shelly 3 EM can measure up to 3 phases. The wiring for this is shown in the following illustration. For the experimental setup, only a single phase was used.

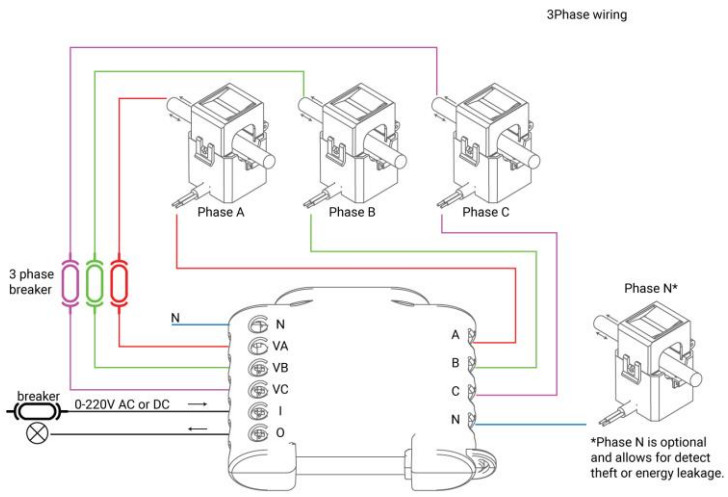


Figure 14 Wiring diagram for the Shelly 3EM for 3-phases (Reference: Doku P5 S.F)

Figure 15 Wiring of the 3EM in the final setup P5 (Reference: Doku P5 S.F)

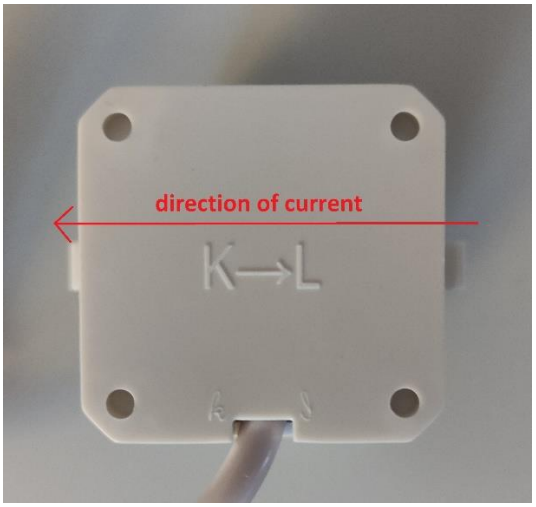


Figure 16 "Current transformer of the 3EM with indicated current direction, so that power is positive (Reference: Doku P5 S.F)

6.7.1.2 Shelly Pro 2PM and Pro 4PM

The test setup is designed so that channel 2 (I2/O2) can be switched. However, since the channels start at zero in the Shelly API, channel 1 must be addressed. This also applies to using the relay with the OpenCEM library.

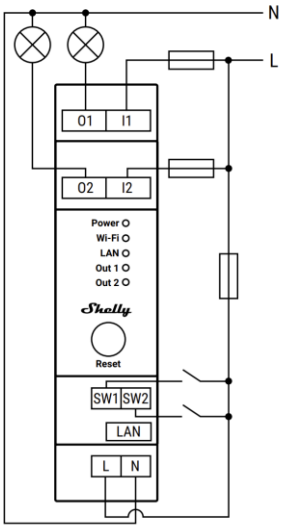


Figure 17 Device diagram of the Shelly Pro 2PM (Reference: Doku P5 S.F)

Figure 18 Wiring of the 2PM in the final setup P5 (Reference: Doku P5 S.F)

.

The relay channel wired for the Pro 4PM is located on O1. It is addressed with channel 0 through the Shelly API and the OpenCEM library.

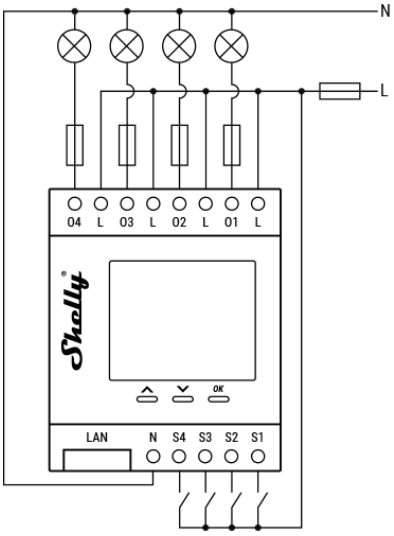


Figure 19 Device diagram of the Shelly Pro 4PM (Reference: Doku P5 S.F)

Figure 20 Wiring of the 4PM in the final setup P5 (Reference: Doku P5 S.F)