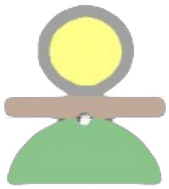


mywattsmon

Manual	
Date:	2024-10-06
Program version:	0.9.5
Author:	berryunit

Content

1. Quickstart

‚mywattsmon‘ is a minimal Python application for monitoring electrical power and energy in the smart home..

- Support for devices such as energy meters, smart plugs, etc.
- 24/7 monitor process with scheduled data storage
- Optional monitor window
- Low resource requirements
- Easily configurable via JSON file
- Extendable with your own device classes

A computer running Python version 3.11 or higher is required. For SBCs such as Raspberry Pi, a hard disk (e.g. a USB SSD) is recommended, as SD cards are generally not suitable for continuous operation.

1.1 Installation

The application should be installed in a user directory, as it saves data and can be extended individually.

```
python -m pip install mywattsmon -U -t <target directory>
```

Alternatively, the release file can be downloaded from the respository and unpacked.

1.2 Usage

In the following, it is assumed that the application has been installed on a Linux computer into the user's home directory (e.g. into `/home/u1/mywattsmon`) and that the calls are made from the home directory (`/home/u1`).

Start the monitor process (exit with Ctrl+C):

```
python -m mywattsmon.app.monitor
```

Start the monitor window (exit with Ctrl+C, in the window using the exit button or escape key):

```
python -m mywattsmon.app.window
```

Note: When the application is started for the first time, the data directory `mywattsmon-data` is created parallel to the application directory. Among other things, this contains the configuration file `config.json` with a configuration of the device class `Mock`. As this class provides random numbers, the application can be executed directly after installation without further configuration.

1.3 Further Information

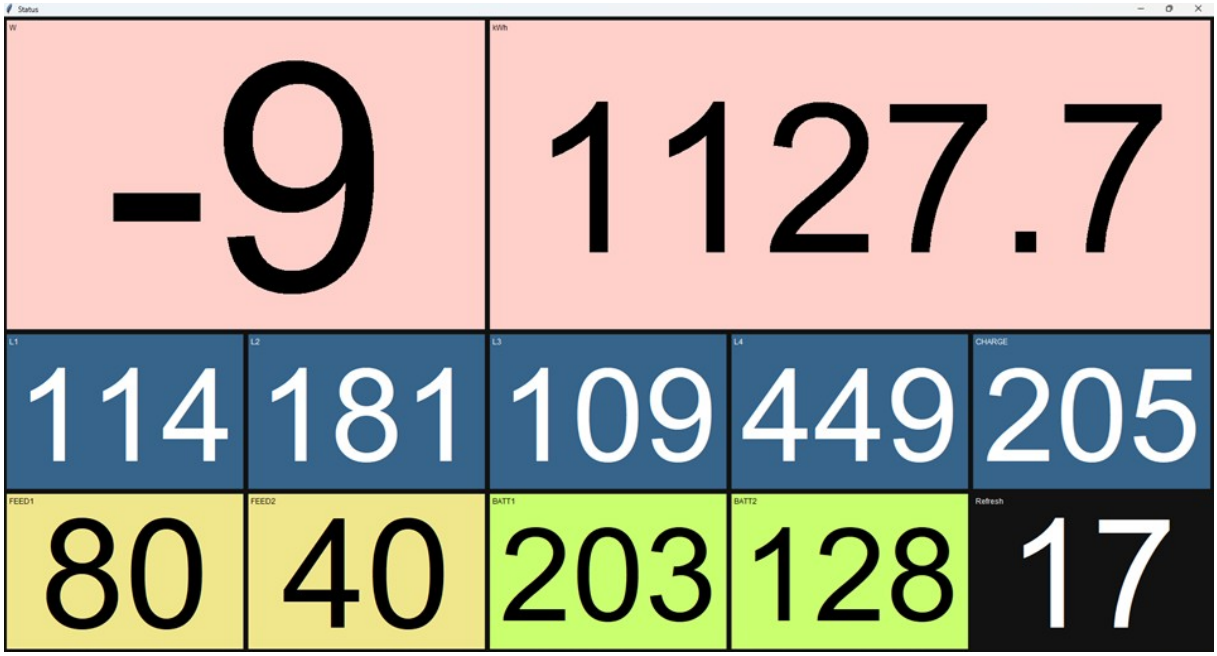
- Documentation: `/mywattsmon/doc/*`
- Repository: `https://github.com/berryunit/mywattsmon`
- License: MIT

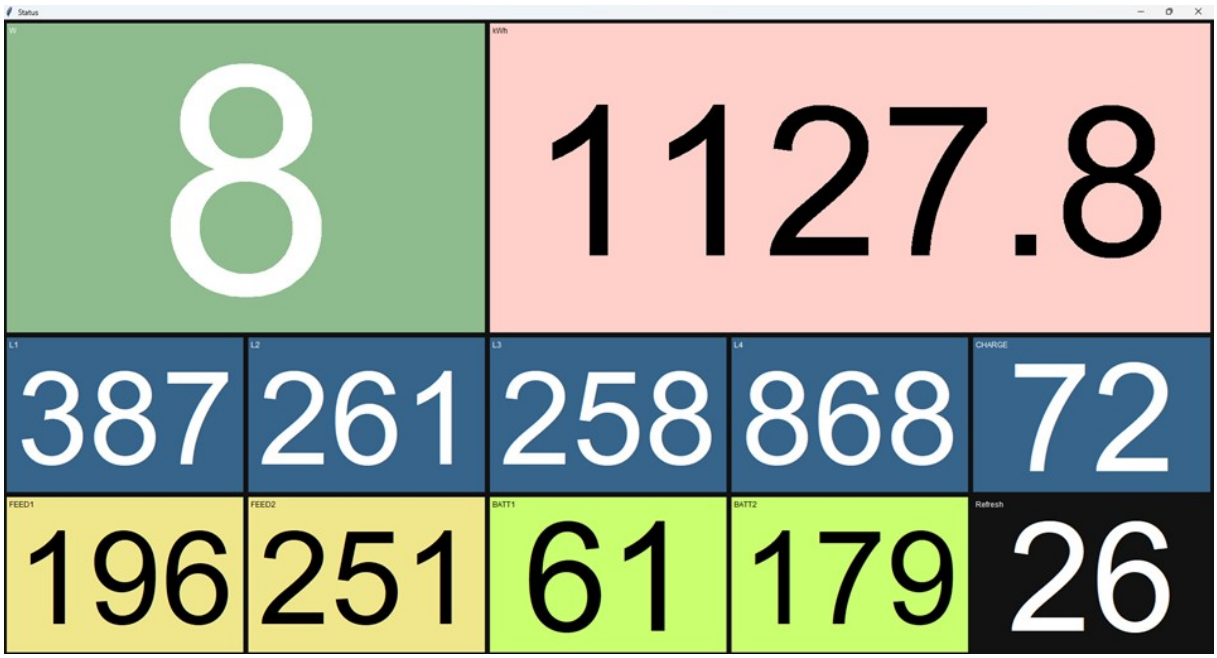
2. Configuration

The following sample images show the monitor window with random numbers of the preconfigured device class `Mock`.

The first line shows the current total power and the cumulative total energy. This simulates a central energy meter that supplies these values in watts respectively kilowatt hours. The color representation of the power value depends on whether it is negative or positive (consumption or feed into the public power grid).

In the field at the bottom right, a countdown runs until the next update of the values. The other fields simulate the power in watts for devices such as smart plugs.





2.1 Overview

First of all, the contents of the file `/mywattsmon-data/config.json` are shown in an overview using an example. The individual configuration areas are then described in detail.

```
{
    "title":"mywattsmon configuration",
    "loglevel":"info",
    "logtofile":"false",
    "port":42001,
    "window":{
        "title":"Status",
        "size":"max",
        "interval":{"default":30,"20:00-22:00":60,"22:00-05:00":300},
        "nightmode":{"timeframe":"22:00-05:00","colors":"ref0"},
        "colors":{
            "window":{"bg":"gray7"},
            "values":{
                "ref0":{"bg":"black","fg":"gray7"},
                <...>,
                "ref6":{"bg":"gray7","fg":"white"}
            }
        },
        "grid":{
            "W":{
                "units":"M1","var":"power","colors":"ref1",
                "rownum":0,"colnum":0,"rowspan":2,"colspan":2
            },
            <...>,
            "Refresh":{
                "units":"<refresh>","var":"<countdown>","colors":"ref6",
                "rownum":3,"colnum":4,"rowspan":1,"colspan":1
            }
        }
    },
    "database":{
        "times":["00:00","03:00","06:00","09:00","12:00","15:00",
                "18:00","21:00","23:59"],
        "tablename":"kwh",
        "columns":{
            "em":{"units":"M1","var":"energy"},
            <...>,
            "batt2":{"units":"M14","var":"energy"}
        }
    },
    "devices":{
        "Mock":{
            "module":"mywattsmon.app.device.mock",
            "units":{
                "M1":{"uid":"m1"},
                <...>,
                "M14":{"uid":"m14"}
            }
        }
    }
}
```

2.2 General Information

Key	Value	Description
`title`	`<title text>`	Informative text for the configuration. Example: `"mywattsmon configuration"`
`loglevel`	`test, info, warning, error`	Level from which information is output to the log target. Example: `"info"`
`logtofile`	`true, false`	Output to a daily log file at `/mywattsmon-data/log/` (`true`) or to the standard output (`false`). Example: `"false"`
`port`	`<port number>`	Local TCP port on which the monitor process waits for local requests from the monitor window. Example: `42001`

2.3 window

Key	Value	Description
`title`	`<title text>`	Informative text for the window frame. Example: `"mywattsmon configuration"`

`size`	`max, full, <w*h+x+y>`	Window size: Largest possible window with frame (`max`), full screen (`full`), or explicit width, height, X and Y position. Examples (alternatively):
		
		`"max"`
		`"full"`
		`"640*400+10+10"`

`interval`	`<list of key-value pairs>`	Specification of time-dependent update intervals. Key: `<Time frame in the format HH:MM-HH:MM>`. Value: `<number of seconds>`. If the current time is outside all specified time frames, the key default is used. Example:
		
		`{`
		`"default":30,`
		`"20:00-22:00":60,`
		`"22:00-05:00":300`
		`}`

`nightmode`	`<list of key-value pairs>`	Specification of a time frame and a color set for a dark display. Key: `timeframe`. Value: `<time frame in the format HH:MM-HH:MM>`. Key: `colors`. Value: `<color reference key>`.

		Example:
		
		{
		"timeframe":"22:00-05:00",
		"colors":"ref0"
		}

colors	<see section 2.3.1 colors>	Specification of the background color and various color sets for the color representation of the values. Colors supported by Python must be specified.
grid	<see section 2.3.2 grid>	Specification of the grid for displaying the values in the window.

2.3.1 colors

Key	Value	Description
window	<key-value pair>	Window background color. Key: bg. Value: <color>. Example: {"bg":"gray7"}

values	<list of key-value pairs>	Specification of reference keys with assignment of a background and foreground color (key bg and fg respectively). If the keys are supplemented with plus or minus sign, the color is set depending on whether the value is positive or negative. Example:
		
		{
		"ref0":{"bg":"black","fg":"gray7"},
		"ref1":{
		"bg+":"darkseagreen","fg+":"white",
		"bg-":"#FFCFC9","fg-":"black"
		
		}
		}

2.3.2 grid

Key	Value	Description
<Label>	<list of key-value pairs>	Grid element specification. A label must be entered for each element and specified as follows: **Key** **Value** units <Device unit names> var <Variable (power or energy)> colors <Color reference key> rownum <Row number (0 bis n)> colnum <Column number (0 bis n)> rowspan <Row count (0 bis n)> colspan <Column count (0 bis n)>

		Examples:
		
		{
		"W":{
		"units":"M1","var":"power","colors":"ref1",
		"rownum":0,"colnum":0,"rowspan":2,"colspan":2
		},
		"kWh":{
		"units":"M1","var":"energy","colors":"ref2",
		"rownum":0,"colnum":2,"rowspan":2,"colspan":3
		},
		"L1":{
		"units":"M2,M3","var":"power","colors":"ref3",
		"rownum":2,"colnum":0,"rowspan":1,"colspan":1
		},
		<...>,
		"Refresh":{
		"units":"***<refresh>***","var":"***<countdown>***",
		"colors":"ref3","rownum":3,"colnum":4,
		"rowspan":1,"colspan":1
		}
		}
		
		Note: In the last example entry, `<refresh>` and `<countdown>` are used to specify the element for displaying the refresh countdown.

2.4 database

Key	Value	Description
`times`	`<list>`	List of times in HH:MM format at which the current values are written to the database. Example: `["00:00","06:00","12:00","18:00","23:59"]` *Note: Enter the times `00:00` and `23:59` to be able to query the data for one day.*
`tablename`	`<tablename>`	Name of the table to be written to in lower case letters. Example: `"kwh"`

`columns`	`<list of key-value pairs>`	Column specification. For each column, a column name compatible with SQLite must be entered in lower case and specified as follows:
		
		Key **Value**
		`units` `<Device unit names>`
		`var` `<Variable (power or energy)>`
		
		Example:
		
		`"em":{"units":"M1","var":"energy},`
		`"l1":{"units":"M2,M3","var":"energy},`
		`<...>,`
		`"batt2":{"units":"M14","var":"energy}`

2.5 devices

Key	Value	Description

`<class name>`	`<list of key-value pairs>`	Device class specification. A simple class name and the explicit module name containing the class must be specified for each device.
		
		For each device unit, the device unit name and its internally required data, for example a unit ID (`uid`), must be specified.
		
		Example for device class `Mock`:
		
		`"Mock":{`
		`"module":"mywattsmon.app.device.mock",`
		`"units":{`
		`"M1":{"uid":"m1"},`
		`"M2":{"uid":"m2"},`
		`<...>,`
		`"M14":{"uid":"m14"}`
		`}`
		`}`
		
		Further information can be found in the following chapter “Device classes”.

2.6 Please note

If changes are made to the configuration, particular attention must be paid:

- The rules specified by the JSON format must be observed. This applies in particular to parentheses, the use of commas and the correct syntax of the specifications.
- If changes are to be made to the database configuration, the previously used database `/mywattsmon-data/db/monitor.db` should first be backed up by copying and then removed. The database and the configured database table are then automatically recreated.

3. Device classes

A device class represents a physical device such as a Fritzbox or a Bosch Smart Home Controller. All device classes supplied follow standards that are described in this chapter and must be taken into account when creating your own device classes.

3.1 Standards

The `mywattsmon.app.device.abstract` module contains the `Abstract` class, which represents an informal Python interface. The interface specifies three methods that are to be overridden by the respective class. The code of the `Abstract` class is shown below, followed by sample code for executing a device class.

```
class Abstract:

    """
    This class is an informal Python interface and acts as a blueprint
    for all device classes. Each device class  implements (overrides)
    its abstract methods.
    """

    def set_config(self, config:dict):
        """Sets the configuration for this device as needed.

        Args:
            config (dict): Specifications from the configuration file.

        Returns:
            None.
        """
        pass

    def close(self):
        """Closes resources as needed.

        Args:
            None.

        Returns:
            None.
        """
        pass

    def get_infoset(self):
        """Gets information from all configured device units.

        The form of the result is standardized. The following data
        must be supplied for each unit:

        data = {}
        data['power'] = 0
        data['energy'] = 0.0
        data['state'] = 'OFF'
        data['code'] = 2
        data['info'] = ''
        data['trace'] = ''

        infoset = {}
        infoset[<unit1>] = <data_from_unit1>
        infoset[<unit2>] = <data_from_unit2>
        infoset[<unit3>] = <data_from_unit3>
        ...

        See also the code of the implemented device classes.

        Args:
            None.

        Returns:
            dict: The data of all units as 'infoset'.
        """
        pass


from mywattsmon.app.device.mock import Mock
instance = Mock()
instance.set_config(…)
while …:
       infoset = instance.get_infoset()
instance.close()
```

How exactly a class is to be configured in detail is determined by the class itself. The following shows an explicit configuration for all supplied device classes that access physical devices. Private attributes such as passwords etc. have been made unrecognizable.

```
"devices":{
    "DTSU":{
        "module":"mywattsmon.app.device.dtsu",
        "units":{
            "EM":{"port":"/dev/ttyUSB2","address":68}
        }
    },
    "VEDirect":{
        "module":"mywattsmon.app.device.vedirect",
        "units":{
            "BATT1":{"port":"/dev/ttyUSB0"},
            "BATT2":{"port":"/dev/ttyUSB1"}
        }
    },
    "Fritz":{
        "module":"mywattsmon.app.device.fritz",
        "connection":{
            "address":"192.168.178.1",
            "user":"fritzuser",
            "password":"…"
        },
        "units":{
            "HEATER":{"uid":"11657 …"},
            "L1":{"uid":"11630 …"},
            "L2":{"uid":"11630 …"},
            "L3":{"uid":"11630 …"},
            "FEED1":{"uid":"11657 …"},
            "FEED2":{"uid":"11657 …"},
            "CHARGE":{"uid":"11657 …"}
        }
    },
    "Bosch":{
        "module":"mywattsmon.app.device.bosch",
        "connection":{
            "address": "192.168.178.27",
            "certfile": "/home/pi/mywattsmon-data/cert/bosch/shccert.pem",
            "keyfile": "/home/pi/mywattsmon-data/cert/bosch/shckey.pem"
        },
        "units":{
            "WASHER":{"uid":"hdm:ZigBee:3425b4fffe…"},
            "COOLER":{"uid":"hdm:ZigBee:70ac08fffe…"},
            "EHOOD":{"uid":"hdm:ZigBee:f4b3b1fffe…"},
            "HEATER1":{"uid":"hdm:ZigBee:f082c0fffe…"},
            "HEATER2":{"uid": "hdm:ZigBee:6c5cb1fffe…"}
        }
    }
}
```

3.2 Custom device classes

Custom device classes must follow the previously documented standards and can be implemented and configured according to the pattern of the supplied classes. As the monitor loads the device classes dynamically, your own device classes should then be directly applicable.

Custom device classes must be stored in the data directory under `/mywattsmon-data/py`. The files are copied from there to `/mywattsmon/app/device/user/` when the application is started and must be configured accordingly as module `mywattsmon.app.device.user.<module name>`.

4. FAQ

1. What does the directory structure of the application look like?

The following table provides information about the directory structure.

Note: Each Python file represents a module with one class. File `/mywattsmon/app/monitor.py`, for example, represents the module `mywattsmon.app.monitor` with the class `Monitor`.

Directory	Content
`mywattsmon`	Application directory.
`/app`	Entry point classes `Monitor` and `Window`.
`/app/device`	Device classes.
`/app/device/user`	Copied custom device classes, if available in the data directory.
`/app/helper`	Helper classes.
`/doc`	Documentation in PDF format, readme file, license file, sample configuration files.
`/test`	Test class (derived from `unittest.TestCase`).
`/test/data`	Test data (created automatically).
`mywattsmon-data`	Data directory with the configuration file `config.json`. Is automatically created parallel to the application directory.
`/db`	Database.
`/log`	Log output.
`/py`	Custom device classes.

2. On which platforms does the application run?

In principle on all platforms on which Python 3.11 or higher runs. Tested with Python 3.11 and 3.12 under Linux and Windows.

3. Does the monitor also work without the monitor window?

Yes, the monitor window is optional and runs in its own process. Communication with the monitor process takes place via socket (local TCP port).

4. Can the application be installed and executed multiple times in parallel?

Yes, but make sure to install and run the application under different directories, for example under `/home/u1/`**`mywm1`**`/mywattsmon` and `/home/u1/`**`mywm2`**`/mywattsmon`. In addition, different port numbers must be configured, for example `42011` and `42012`.

5. Can an alternative data directory be specified when calling up the monitor or the monitor window?

Yes, you must use the `-d` or `--datapath` option when calling it. Make sure to specify the same data directory for both calls.

6. Can multiple databases and/or database tables be configured?

No. The database `/mywattsmon-data/db/monitor.db` and the configured database table are always used. If the database configuration is changed, the database must first be backed up by copying and then removed. It is then automatically recreated with the currently configured database table.

7. How can the data be read from the database?

It is an SQLite database that can be read with any compatible SQL client. The time stamp automatically created in the configured database table allows a timeframe to be set when selecting the data records, and the numerical data fields can be used to perform calculations via SQL. This makes it very easy to generate individual statistics. If such an evaluation of the data is planned, copies of `/mywattsmon-data/db/monitor.db` should be made regularly to back up the data.

To illustrate data evaluation, here are some queries with the SQLite CLI tool:

```
$ sqlite3 "/home/u1/mywattsmon-data/db/monitor.db"
sqlite> select max(id) from kwh;
280
sqlite> select id, ts, em from kwh where id = 280;
280|2024-10-06 09:00|1794.7
sqlite> select round(max(em)-min(em),1) from kwh where ts like '2024-09-25%';
2.1
sqlite> select round(max(em)-min(em),1) from kwh where ts >= '2024-09-23' and
ts <= '2024-09-29';
11.4
sqlite> select round(max(em)-min(em),1) from kwh where ts like '2024-09%';
34.3
sqlite> .quit
```

8. Can the supplied productive device classes be used directly?

The productive device classes each require a physical device such as an energy meter or a smart plug. In addition, these classes may have Python dependencies that require additional installation (e.g. `requests`, `minimalmodbus` or `fritzconnection`. See the source code of the classes under `/mywattsmon/app/device/`). If the prerequisites are met, the configuration can be adapted accordingly and the device class can be used.

9. What happens to the own data and custom device classes when the application is updated?

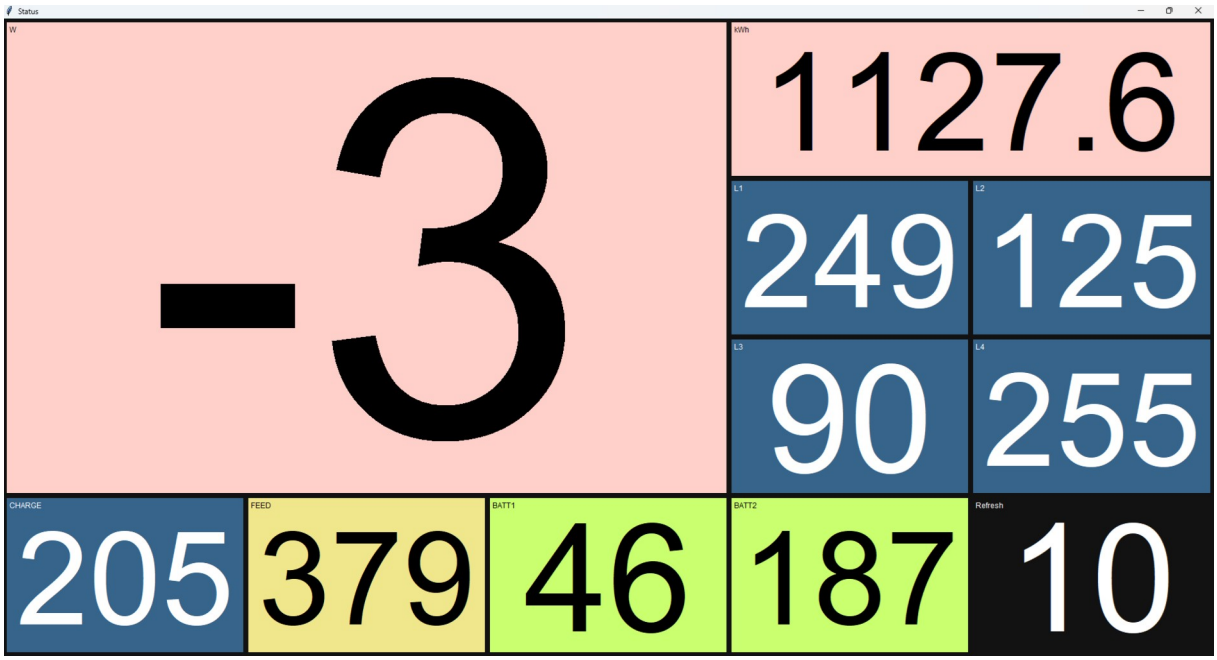
An update with `python -m pip install mywattsmon -U -t <target directory>` updates the entire application directory with all its contents. The data directory `/mywattsmon-data`, which is installed in parallel to the application directory by default, is not affected by this and will not be overwritten the next time the application is started.

The custom device classes located in the data subdirectory `/mywattsmon-data/py` are copied to `/mywattsmon/app/device/user` and loaded dynamically when the application is started. They can therefore also be used again after updating the application.

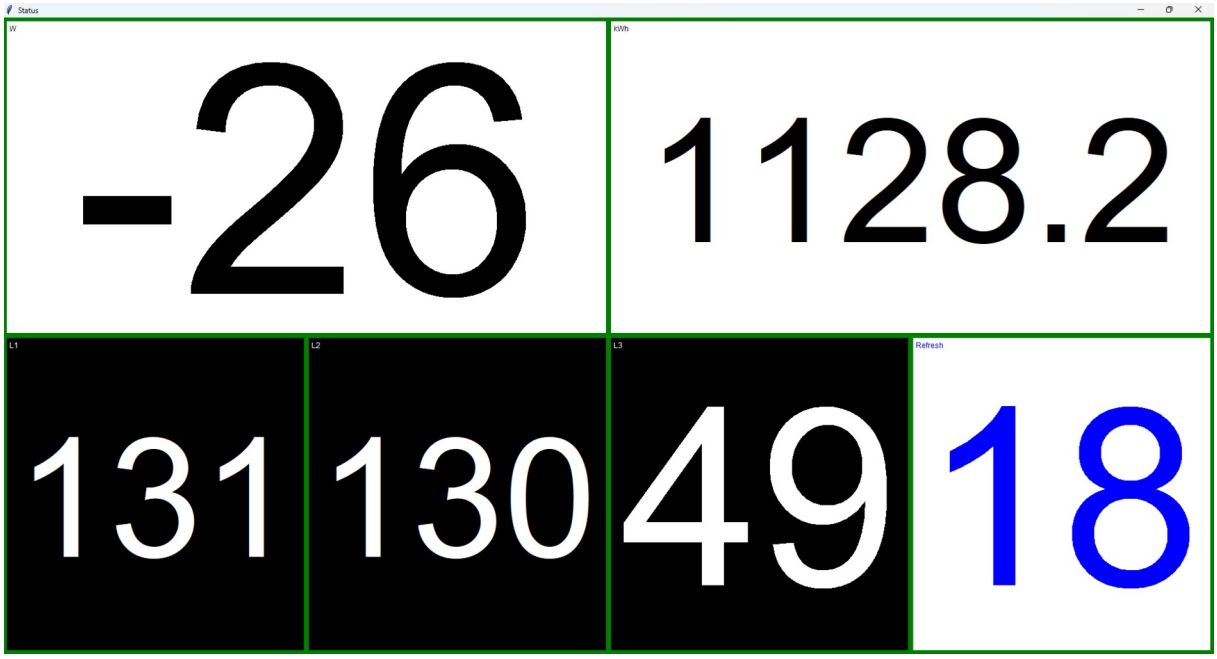
10. Which configuration examples are supplied?

A complete configuration file with all the supplied productive device classes, as well as sample files in which the `Mock` class and a sample `UserMock` class are configured, are available under `/mywattsmon/doc/`. Private attributes such as passwords etc. are made unrecognizable.

Window view with the variant `sample_mock_2_config.json`:



Window view with the variant `sample_usermock_config.json`:



Note: The example device class configured in this variant is coded in the file `usermock.py`, which is also located under `/mywattsmon/doc/`.

11. What is the difference between device and device unit?

In this context, a device is a physical device such as a Fritzbox, a Bosch Smart Home Controller or a Victron Energy Charge Controller. A device class represents a physical device. A device unit is, for example, a smart plug that is connected to the Fritzbox via DECT or to the Bosch controller via ZigBee. In the case of the Victron Energy Charge Controller, device unit and device are practically the same.

12. What value is output if several device unit names are specified in the window grid or database configuration for `units`?

The values of the specified device units are added (`power` in watts, `energy` in kilowatt hours).

13. What do special characters or letters in the window grid mean?

If a grid element does not contain a number but a special character or a letter, these have the following meaning:

Character	Meaning
–	No value (None/null)
~	The device unit is OFF
v	Invalid value
w	Warning (see Log)
x	Error (see Log)

14. Can the monitor also be accessed remotely?

This is not intended. In principle, this would be possible, but the application is designed for local operation.