bluetooth-clocks Documentation

Release 0.2.0.post1.dev2+g5085e7a

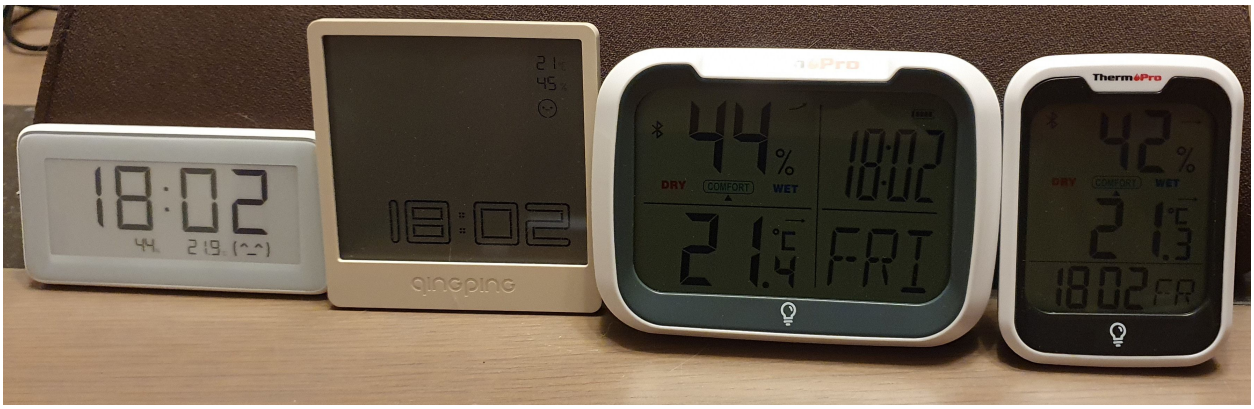
Koen Vervloesem

Oct 28, 2023

CONTENTS

Set and get the time on various Bluetooth Low Energy clocks

This project offers a way to easily recognize Bluetooth Low Energy (BLE) clocks from their advertisements and has a device-independent API to set and get the time on them.



CHAPTER

ONE

CONTENTS

1.1 Installation and usage

1.1.1 Installation

You can install bluetooth-clocks as a package from PyPI with pip:

```
pip install bluetooth-clocks
```

1.1.2 Usage of the command-line program

If you have installed the package with `pip`, you can run the program as `bluetooth-clocks`:

```
$ bluetooth-clocks -h
usage: bluetooth-clocks [-h] [--version] [-v] [-vv] {discover,get,set} ...

Bluetooth Clocks

options:
  -h, --help           show this help message and exit
  --version            show program's version number and exit
  -v, --verbose        set loglevel to INFO
  -vv, --very-verbose  set loglevel to DEBUG

Subcommands:
  {discover,get,set}
    discover           discover supported Bluetooth clocks
    get                get the time from a Bluetooth clock
    set                set the time of a Bluetooth clock
```

Discovering Bluetooth clocks

You can discover supported Bluetooth clocks with `bluetooth-clocks discover`:

```
$ bluetooth-clocks discover
Scanning for supported clocks...
Found a ThermoPro TP358: address BC:C7:DA:6A:52:C6, name TP358 (52C6)
Found a Xiaomi LYWSD02: address E7:2E:00:B1:38:96, name LYWSD02
Found a ThermoPro TP393: address 10:76:36:14:2A:3D, name TP393 (2A3D)
Found a Qingping BT Clock Lite: address 58:2D:34:54:2D:2C, name Qingping BT Clock Lite
Found a Current Time Service: address EB:76:55:B9:56:18, name F15
```

These are the options that the `discover` subcommand recognizes:

```
$ bluetooth-clocks discover -h
usage: bluetooth-clocks discover [-h] [-s SCAN_DURATION]

options:
  -h, --help            show this help message and exit
  -s SCAN_DURATION, --scan-duration SCAN_DURATION
                        scan duration (default: 5 seconds)
```

Setting the time

Set the time of a clock with a given Bluetooth address:

```
$ bluetooth-clocks set -a E7:2E:00:B1:38:96
Scanning for device E7:2E:00:B1:38:96...
Writing time to device...
Synchronized time
```

If you want to regularly synchronize the time on the device, you can run this command as a service, e.g. with a systemd service or in a cron job in Linux.

These are the options that the `set` subcommand recognizes:

```
$ bluetooth-clocks set -h
usage: bluetooth-clocks set [-h] -a ADDRESS [-s SCAN_DURATION] [-t TIME] [-p]

options:
  -h, --help            show this help message and exit
  -a ADDRESS, --address ADDRESS
                        Bluetooth address (e.g. 12:34:56:78:9A:BC)
  -s SCAN_DURATION, --scan-duration SCAN_DURATION
                        scan duration (default: 5 seconds)
  -t TIME, --time TIME  the time to set, in ISO 8601 format (e.g. 2023-01-10T16:20,
                        default: current time)
  -p, --am-pm           use AM/PM format (default: 24-hour format)
```

Warning: Don't be a jerk by changing the time of other people's clocks. Use this tool responsibly.

Getting the time

Get the time from a clock with a given Bluetooth address:

```
$ bluetooth-clocks get -a E7:2E:00:B1:38:96
Scanning for device E7:2E:00:B1:38:96...
Reading time from device...
2023-01-14T17:54:17
```

These are the options that the `get` subcommand recognizes:

```
$ bluetooth-clocks get -h
usage: bluetooth-clocks get [-h] -a ADDRESS [-s SCAN_DURATION]

options:
  -h, --help            show this help message and exit
  -a ADDRESS, --address ADDRESS
                        Bluetooth address (e.g. 12:34:56:78:9A:BC)
  -s SCAN_DURATION, --scan-duration SCAN_DURATION
                        scan duration (default: 5 seconds)
```

1.1.3 Usage of the library

The functionality of the command-line program can also be used in your own Python programs by using this project as a library.

See the module reference for complete API documentation.

1.2 Supported devices

Bluetooth Clocks supports the following devices:

Device	Set time	Set 12/24h format	Read time
Current Time Service (e.g. PineTime with InfiniTime firmware)	Yes	No	Yes
PVVX firmware (LYWSD03MMC, MHO-C401, CGG1, CGDK2, MJWSD05MMC, MHO-C122)	Yes	No	Yes
Qingping BT Clock Lite	Yes	No	No
ThermoPro TP358/TP393	Yes	Yes	No
Xiaomi LYWSD02	Yes	No	Yes

1.3 Contributing

Welcome to `bluetooth-clocks` contributor's guide.

This document focuses on getting any potential contributor familiarized with the development processes, but other kinds of contributions are also appreciated.

If you are new to using git or have never collaborated in a project previously, please have a look at contribution-guide.org. Other resources are also listed in the excellent guide created by FreeCodeCamp[1].

Please notice, all users and contributors are expected to be **open, considerate, reasonable, and respectful**. When in doubt, Python Software Foundation's Code of Conduct is a good reference in terms of behavior guidelines.

1.3.1 Issue Reports

If you experience bugs or general issues with `bluetooth-clocks`, please have a look on the issue tracker. If you don't see anything useful there, please feel free to fire an issue report.

Tip: Please don't forget to include the closed issues in your search. Sometimes a solution was already reported, and the problem is considered **solved**.

New issue reports should include information about your programming environment (e.g., operating system, Python version) and steps to reproduce the problem. Please try also to simplify the reproduction steps to a very minimal example that still illustrates the problem you are facing. By removing other factors, you help us to identify the root cause of the issue.

1.3.2 Documentation Improvements

You can help improve `bluetooth-clocks` docs by making them more readable and coherent, or by adding missing information and correcting mistakes.

`bluetooth-clocks` documentation uses Sphinx as its main documentation compiler. This means that the docs are kept in the same repository as the project code, and that any documentation update is done in the same way was a code contribution.

The documentation is written in the reStructuredText markup language.

Tip: Please notice that the GitHub web interface provides a quick way of propose changes in `bluetooth-clocks`'s files. While this mechanism can be tricky for normal code contributions, it works perfectly fine for contributing to the docs, and can be quite handy.

If you are interested in trying this method out, please navigate to the `docs` folder in the source repository, find which file you would like to propose changes and click in the little pencil icon at the top, to open GitHub's code editor. Once you finish editing the file, please write a message in the form at the bottom of the page describing which changes have you made and what are the motivations behind them and submit your proposal.

When working on documentation changes in your local machine, you can compile them using `tox`:

```
tox -e docs
```

[1] Even though, these resources focus on open source projects and communities, the general ideas behind collaborating with other developers to collectively create software are general and can be applied to all sorts of environments, including private companies and proprietary code bases.

and use Python's built-in web server for a preview in your web browser (`http://localhost:8000`):

```
python3 -m http.server --directory 'docs/_build/html'
```

1.3.3 Code Contributions

See the module reference for complete documentation of this library.

If you want to add support for a new device, you need to create a subclass of the `BluetoothClock` class. Have a look at the existing classes for devices.

Submit an issue

Before you work on any non-trivial code contribution it's best to first create a report in the issue tracker to start a discussion on the subject. This often provides additional considerations and avoids unnecessary work.

Create an environment

Before you start coding, we recommend creating an isolated virtual environment to avoid any problems with your installed Python packages. This can easily be done via either `virtualenv`:

```
virtualenv <PATH TO VENV>
source <PATH TO VENV>/bin/activate
```

or Miniconda:

```
conda create -n bluetooth-clocks python=3 six virtualenv pytest pytest-cov
conda activate bluetooth-clocks
```

Clone the repository

1. Create an user account on GitHub if you do not already have one.
2. Fork the project repository: click on the *Fork* button near the top of the page. This creates a copy of the code under your account on GitHub.
3. Clone this copy to your local disk:

   ```
   git clone git@github.com:YourLogin/bluetooth-clocks.git
   cd bluetooth-clocks
   ```

4. You should run:

   ```
   pip install -U pip setuptools -e .
   ```

   to be able to import the package under development in the Python REPL.

5. Install `pre-commit`:

   ```
   pip install pre-commit
   pre-commit install
   ```

   `bluetooth-clocks` comes with a lot of hooks configured to automatically help the developer to check the code being written.

Implement your changes

1. Create a branch to hold your changes:

   ```
   git checkout -b my-feature
   ```

   and start making changes. Never work on the main branch!

2. Start your work on this branch. Don't forget to add docstrings to new functions, modules and classes, especially if they are part of public APIs.

3. Add yourself to the list of contributors in `AUTHORS.rst`.

4. When you're done editing, do:

   ```
   git add <MODIFIED FILES>
   git commit
   ```

   to record your changes in git.

   Please make sure to see the validation messages from `pre-commit` and fix any eventual issues. This should automatically use flake8/black to check/fix the code style in a way that is compatible with the project.

   Important: Don't forget to add unit tests and documentation in case your contribution adds an additional feature and is not just a bugfix.

   Moreover, writing a descriptive commit message is highly recommended. In case of doubt, you can check the commit history with:

   ```
   git log --graph --decorate --pretty=oneline --abbrev-commit --all
   ```

   to look for recurring communication patterns.

   ---

5. Please check that your changes don't break any unit tests with:

   ```
   tox
   ```

   (after having installed `tox` with `pip install tox` or `pipx`).

   You can also use `tox` to run several other pre-configured tasks in the repository. Try `tox -av` to see a list of the available checks.

Submit your contribution

1. If everything works fine, push your local branch to GitHub with:

   ```
   git push -u origin my-feature
   ```

2. Go to the web page of your fork and click "Create pull request" to send your changes for review.

   Find more detailed information in creating a PR. You might also want to open the PR as a draft first and mark it as ready for review after the feedbacks from the continuous integration (CI) system or any required fixes.

Troubleshooting

The following tips can be used when facing problems to build or test the package:

1. Make sure to fetch all the tags from the upstream repository. The command `git describe --abbrev=0 --tags` should return the version you are expecting. If you are trying to run CI scripts in a fork repository, make sure to push all the tags. You can also try to remove all the egg files or the complete egg folder, i.e., `.eggs`, as well as the `*.egg-info` folders in the `src` folder or potentially in the root of your project.

2. Sometimes `tox` misses out when new dependencies are added, especially to `setup.cfg` and `docs/requirements.txt`. If you find any problems with missing dependencies when running a command with `tox`, try to recreate the `tox` environment using the `-r` flag. For example, instead of:

   ```
   tox -e docs
   ```

   Try running:

   ```
   tox -r -e docs
   ```

3. Make sure to have a reliable `tox` installation that uses the correct Python version (e.g., 3.8+). When in doubt you can run:

   ```
   tox --version
   # OR
   which tox
   ```

   If you have trouble and are seeing weird errors upon running `tox`, you can also try to create a dedicated virtual environment with a `tox` binary freshly installed. For example:

   ```
   virtualenv .venv
   source .venv/bin/activate
   .venv/bin/pip install tox
   .venv/bin/tox -e all
   ```

4. Pytest can drop you in an interactive session in the case an error occurs. In order to do that you need to pass a `--pdb` option (for example by running `tox -- -k <NAME OF THE FALLING TEST> --pdb`). You can also setup breakpoints manually instead of using the `--pdb` option.

1.3.4 Maintainer tasks

Releases

If you are part of the group of maintainers and have correct user permissions on PyPI, the following steps can be used to release a new version for `bluetooth-clocks`:

1. Make sure all unit tests are successful.
2. Tag the current commit on the main branch with a release tag, e.g., `v1.2.3`.
3. Push the new tag to the upstream repository, e.g., `git push upstream v1.2.3`
4. Clean up the `dist` and `build` folders with `tox -e clean` (or `rm -rf dist build`) to avoid confusion with old builds and Sphinx docs.
5. Run `tox -e build` and check that the files in `dist` have the correct version (no `.dirty` or git hash) according to the git tag. Also check the sizes of the distributions, if they are too big (e.g., > 500KB), unwanted clutter may have been accidentally included.

6. Run `tox -e publish -- --repository pypi` and check that everything was uploaded to PyPI correctly.

1.4 License

1.5 Contributors

- Koen Vervloesem

1.6 Changelog

1.6.1 Version 0.2.0: Get time from PVVX (2023-07-04)

This release adds support for reading the time from devices with PVVX firmware.

- Migrate code linting to Ruff, apply fixes by @koenvervloesem in https://github.com/koenvervloesem/bluetooth-clocks/pull/15
- Add MJWSD05MMC and MHO-C122 to list of supported PVVX models by @koenvervloesem in https://github.com/koenvervloesem/bluetooth-clocks/pull/16
- Assume if TYPE_CHECKING is covered by @koenvervloesem in https://github.com/koenvervloesem/bluetooth-clocks/pull/17
- Deprecate Python 3.7 by @koenvervloesem in https://github.com/koenvervloesem/bluetooth-clocks/pull/18
- Add read time command for PVVX firmware by @koenvervloesem in https://github.com/koenvervloesem/bluetooth-clocks/pull/19

1.6.2 Version 0.1.2: Local time, please (2023-02-02)

This is a bugfix release. Previously the time on Qingping devices and devices running the PVVX ATC firmware was set to UTC instead of local time.

- Autoupdate pre-commit by @koenvervloesem in https://github.com/koenvervloesem/bluetooth-clocks/pull/12
- Fix local time on PVVX and Qingping devices by @koenvervloesem in https://github.com/koenvervloesem/bluetooth-clocks/pull/13

1.6.3 Version 0.1.1 (2023-01-23)

This is a bugfix release.

- Fix doctest: use UTC in get_time_from_bytes example by @koenvervloesem in https://github.com/koenvervloesem/bluetooth-clocks/pull/3
- Fix link to Bleak's BLEDevice in docs by @koenvervloesem in https://github.com/koenvervloesem/bluetooth-clocks/pull/4
- Add codecov badge to README by @koenvervloesem in https://github.com/koenvervloesem/bluetooth-clocks/pull/5
- Various documentation fixes by @koenvervloesem in https://github.com/koenvervloesem/bluetooth-clocks/pull/6
- Build documentation on Read The Docs with Python 3.11 by @koenvervloesem in https://github.com/koenvervloesem/bluetooth-clocks/pull/7
- Use Ubuntu 22.04/Python 3.11 for Read The Docs by @koenvervloesem in https://github.com/koenvervloesem/bluetooth-clocks/pull/8
- Update to PyScaffold v4.4 project features by @koenvervloesem in https://github.com/koenvervloesem/bluetooth-clocks/pull/9
- Import future annotations by @koenvervloesem in https://github.com/koenvervloesem/bluetooth-clocks/pull/10

1.6.4 Version 0.1.0 (2023-01-20)

Initial version of Bluetooth Clocks

1.7 bluetooth_clocks

1.7.1 bluetooth_clocks package

Set and get the time on various Bluetooth clocks.

This project offers a way to easily recognize Bluetooth Low Energy clocks from their advertisements and has a device-independent API to set and get the time on them.

class bluetooth_clocks.**BluetoothClock**(*device: BLEDevice*)

Bases: ABC

Abstract class that represents the definition of a Bluetooth clock.

Support for every type of Bluetooth clock is implemented as a separate subclass by giving the class variables a value and/or by overriding methods or implementing abstract methods of this class.

address

The Bluetooth address of the device.

Type
str

name

The name of the device, or `None` if it doesn't have a name.

Type
str | None

CHAR_UUID: ClassVar[UUID]

The UUID of the characteristic used to read/write the time.

DEVICE_TYPE: ClassVar[str]

The name of the device type.

LOCAL_NAME: ClassVar[str | None]

The local name used to recognize this type of device.

This is `None` if the local name isn't used to recognize the device.

LOCAL_NAME_STARTS_WITH: ClassVar[bool | None]

Whether the local name should start with *LOCAL_NAME*.

`True` if the start of *LOCAL_NAME* is used to recognize this type of device. `False` if the local name should exactly match *LOCAL_NAME*. This is `None` if the local name isn't used to recognize the device.

SERVICE_UUID: ClassVar[UUID]

The UUID of the service used to read/write the time.

TIME_GET_FORMAT: ClassVar[str | None]

The format string to convert bytes read from the device to a time.

This is `None` if the device doesn't support reading the time.

TIME_SET_FORMAT: ClassVar[str]

The format string to convert a time to bytes written to the device.

WRITE_WITH_RESPONSE: ClassVar[bool]

`True` if the bytes to set the time should use write with response.

classmethod create_from_advertisement(*device: BLEDevice*, *advertisement_data: AdvertisementData*) → *BluetoothClock*

Create object of a *BluetoothClock* subclass from advertisement data.

This is a factory method that you use if you don't know the exact device type beforehand. This method automatically recognizes the device type and creates an object of the corresponding subclass.

Parameters

- **device** (*BLEDevice*) – The Bluetooth device.
- **advertisement_data** (*AdvertisementData*) – The advertisement data.

Raises
UnsupportedDeviceError – If the device with address *address* isn't supported.

Returns
An object of the subclass corresponding to the recognized device type.

Return type
BluetoothClock

abstract get_bytes_from_time(*timestamp: float*, *ampm: bool = False*) → bytes

Generate the bytes to set the time on this device.

Override this method in a subclass to implement the device's time format.

Parameters

- **timestamp** (*float*) – The time encoded as a Unix timestamp.
- **ampm** (*bool*) – `True` if the device should show the time with AM/PM, `False` if it should use 24-hour format. Devices that don't support choosing the mode can ignore this argument.

Returns
The bytes needed to set the time of the device to *timestamp*.

Return type
bytes

Example

```
>>> from bluetooth_clocks.devices.thermopro import TP393
>>> from bleak.backends.device import BLEDevice
>>> from datetime import datetime
>>> clock = TP393(BLEDevice("10:76:36:14:2A:3D", "TP393 (2A3D)", {}, -67))
>>> timestamp = datetime.fromisoformat("2023-01-07 17:32:50").timestamp()
>>> clock.get_bytes_from_time(timestamp, ampm=True).hex()
'a5170107112032060005a'
```

async get_time() → float

Get the time of the Bluetooth clock.

Raises
TimeNotReadableError – If the device doesn't support getting the time.

Returns
The time of the Bluetooth clock.

Return type
float

get_time_from_bytes(*time_bytes: bytes*) → float

Convert bytes read from a device to a timestamp.

Override this method in a subclass for a device that supports getting the time.

Parameters
time_bytes (*bytes*) – The raw bytes read from the device.

Raises

- ***InvalidTimeBytesError*** – If *time_bytes* don't have the right format.
- ***TimeNotReadableError*** – If the device doesn't support getting the time.

Returns
The time encoded as a Unix timestamp.

Return type
float

Example

```
>>> from bluetooth_clocks.devices.xiaomi import LYWSD02
>>> from bleak.backends.device import BLEDevice
>>> from datetime import datetime
>>> clock = LYWSD02(BLEDevice("E7:2E:00:B1:38:96", "", {}, -67))
>>> timestamp = clock.get_time_from_bytes(
...             bytes([0xdd, 0xbc, 0xb9, 0x63, 0x00]))
>>> print(datetime.utcfromtimestamp(timestamp).strftime("%Y-%m-%d %H:%M:%S"))
2023-01-07 18:41:33
```

classmethod is_readable() → bool

Test whether you can read the time from this device.

Returns
`True` if this device supports reading the time, `False` otherwise.

Return type
bool

Example

```
>>> from bluetooth_clocks.devices.xiaomi import LYWSD02
>>> from bluetooth_clocks.devices.qingping import CGC1
>>> LYWSD02.is_readable()
True
>>> CGC1.is_readable()
False
```

classmethod recognize(*device: BLEDevice*, *advertisement_data: AdvertisementData*) → bool

Recognize this device type from advertisement data.

By default this checks whether the advertisement data has a local name that is equal to or starts with *LOCAL_NAME*, by calling *recognize_from_local_name()*.

Override this method in a subclass if the device type should be recognized in another way from advertisement data.

Parameters

- **device** (*BLEDevice*) – The Bluetooth device.
- **advertisement_data** (*AdvertisementData*) – The advertisement data.

Returns
`True` if this subclass of *BluetoothClock* recognizes the device, `False` otherwise.

Return type
bool

classmethod recognize_from_local_name(*local_name: str | None*) → bool

Recognize the device from an advertised local name.

This is a helper method that subclasses can use to implement their *recognize()* method.

Parameters
local_name (*str | None = None*) – The local name of the device, or `None` if it doesn't advertise its local name.

Returns
`True` if this subclass of *BluetoothClock* recognizes the device from its local name *local_name*, `False` otherwise.

Return type
bool

classmethod recognize_from_service_uuids(*service_uuids: list[str] | None*) → bool

Recognize this device type from service UUIDs.

This is a helper method that subclasses can use to implement their *recognize()* method.

Parameters
service_uuids (*list[str] | None = None*) – Service UUIDs of the device, or `None` if the device doesn't advertise service UUIDs.

Returns
`True` if this subclass of *BluetoothClock* recognizes the device from the service UUIDs in *service_uuids*, `False` otherwise.

Return type
bool

async set_time(*timestamp: float | None = None*, *ampm: bool = False*) → None

Set the time of the Bluetooth clock.

Parameters

- **timestamp** (*float | None = None*) – The timestamp to write to the clock. If this is `None`, the current time is used.
- **ampm** (*bool*) – `True` if the device should show the time with AM/PM, `False` if it should use 24-hour format. Devices that don't support choosing the mode can ignore this argument.

bluetooth_clocks.**MICROSECONDS = 1000000**

The number of microseconds in a second.

You can use this constant in subclasses of *BluetoothClock*.

bluetooth_clocks.**SECONDS_IN_HOUR = 3600**

The number of seconds in an hour.

You can use this constant in subclasses of *BluetoothClock*.

bluetooth_clocks.**supported_devices**() → list[str]

Get a list of names of supported devices.

Returns
A list of the names of devices supported by this library.

Return type
list[str]

Example

```
>>> from bluetooth_clocks import supported_devices
>>> "ThermoPro TP393" in supported_devices()
True
```

Subpackages

bluetooth_clocks.devices package

Device-specific Bluetooth clock support.

Each submodule in this package implements support for a specific type of Bluetooth clock.

Each class in these submodules implements support for a specific model Bluetooth clock.

For instance, the *bluetooth_clocks.devices.thermopro* module has classes *bluetooth_clocks.devices.thermopro.TP358* and *bluetooth_clocks.devices.thermopro.TP393* for the ThermoPro TP358 and TP393, respectively.

Submodules

bluetooth_clocks.devices.current_time_service module

Bluetooth clock support for devices implementing the Current Time Service.

This includes the PineTime with InfiniTime firmware.

class bluetooth_clocks.devices.current_time_service.**CurrentTimeService**(*device: BLEDevice*)

Bases: *BluetoothClock*

Bluetooth clock support for devices implementing the Current Time Service.

This implements the standardized Bluetooth service Current Time Service (https://www.bluetooth.com/specifications/specs/current-time-service-1-1/).

CHAR_UUID: ClassVar[UUID] = UUID('00002a2b-0000-1000-8000-00805f9b34fb')

The UUID of the characteristic used to read/write the time.

DEVICE_TYPE: ClassVar[str] = 'Current Time Service'

The name of the device type.

SERVICE_UUID: ClassVar[UUID] = UUID('00001805-0000-1000-8000-00805f9b34fb')

The UUID of the service used to read/write the time.

TIME_GET_FORMAT: ClassVar[str | None] = '<HBBBBBBB'

The format string to convert bytes read from the Current Time Service to a time.

This starts with an unsigned short in little-endian format, followed by seven bytes.

TIME_SET_FORMAT: ClassVar[str] = '<HBBBBBBBB'

The format string to convert a time to bytes written to the device.

This starts with an unsigned short in little-endian format, followed by eight bytes.

WRITE_WITH_RESPONSE: ClassVar[bool] = True

Writing the time to the Current Time Service needs write with response.

get_bytes_from_time(*timestamp: float*, *ampm: bool = False*) → bytes

Generate the bytes to set the time on the Current Time Service.

Parameters

- **timestamp** (*float*) – The time encoded as a Unix timestamp.
- **ampm** (*bool*) – `True` if the device should show the time with AM/PM, `False` if it should use 24-hour format. The Current Time Service ignores this argument, as it doesn't support this option.

Returns

The bytes needed to set the time of the device to *timestamp*.

Return type

bytes

get_time_from_bytes(*time_bytes: bytes*) → float

Convert bytes read from the Current Time Service to a timestamp.

Parameters

time_bytes (*bytes*) – The raw bytes read from the device.

Raises

InvalidTimeBytesError – If *time_bytes* don't have the right format.

Returns

The time encoded as a Unix timestamp.

Return type

float

classmethod recognize(*device: BLEDevice*, *advertisement_data: AdvertisementData*) → bool

Recognize the Current Time Service from advertisement data.

This checks whether the Current Time Service's service UUID is in the list of advertised service UUIDs.

Parameters

- **device** (*BLEDevice*) – The Bluetooth device.
- **advertisement_data** (*AdvertisementData*) – The advertisement data.

Returns

`True` if the device is recognized as a Current Time Service, `False` otherwise.

Return type

bool

class bluetooth_clocks.devices.current_time_service.**InfiniTime**(*device: BLEDevice*)

Bases: *CurrentTimeService*

Bluetooth clock support for the PineTime with InfiniTime firmware.

DEVICE_TYPE: ClassVar[str] = 'InfiniTime'

The name of the device type.

LOCAL_NAME: ClassVar[str | None] = 'InfiniTime'

The local name used to recognize this type of device.

LOCAL_NAME_STARTS_WITH: ClassVar[bool | None] = False

The local name should exactly match *LOCAL_NAME*.

classmethod recognize(*device: BLEDevice*, *advertisement_data: AdvertisementData*) → bool

Recognize the PineTime with InfiniTime firmware from advertisement data.

This checks whether the advertisement data has a local name that is equal to or starts with *LOCAL_NAME*.

Parameters

- **device** (*BLEDevice*) – The Bluetooth device.
- **advertisement_data** (*AdvertisementData*) – The advertisement data.

Returns

`True` if the device is recognized as a PineTime with InfiniTime firmware, `False` otherwise.

Return type

bool

bluetooth_clocks.devices.pvvx module

Bluetooth clock support for devices running the PVVX firmware.

class bluetooth_clocks.devices.pvvx.**PVVX**(*device: BLEDevice*)

Bases: *BluetoothClock*

Bluetooth clock support for devices running the PVVX firmware.

CHAR_UUID: ClassVar[UUID] = UUID('00001f1f-0000-1000-8000-00805f9b34fb')

The UUID of the characteristic used to read/write the time.

DEVICE_TYPE: ClassVar[str] = 'PVVX'

The name of the device type.

PVVX_GET_SET_TIME_COMMAND = 35

Command for PVVX firmware to Get/Set Time.

SERVICE_DATA_UUID = UUID('0000181a-0000-1000-8000-00805f9b34fb')

UUID of the service data the PVVX device is advertising.

SERVICE_UUID: ClassVar[UUID] = UUID('00001f10-0000-1000-8000-00805f9b34fb')

The UUID of the service used to read/write the time.

TIME_GET_FORMAT: ClassVar[str | None] = '<BLL'

The format string to convert bytes read from the device to a time.

TIME_SET_FORMAT: ClassVar[str] = '<BL'

The format string to convert a time to bytes written to the PVVX device.

WRITE_WITH_RESPONSE: ClassVar[bool] = False

Writing the time to the PVVX device needs write without response.

get_bytes_from_time(*timestamp: float*, *ampm: bool = False*) → bytes

Generate the bytes to set the time on the PVVX device.

Parameters

- **timestamp** (*float*) – The time encoded as a Unix timestamp.

- **ampm** (*bool*) – `True` if the device should show the time with AM/PM, `False` if it should use 24-hour format. The PVVX device ignores this argument, as it doesn't support this option.

Returns
The bytes needed to set the time of the device to *timestamp*.

Return type
bytes

async get_time() → float

Get the time of the PVVX device.

Returns
The time of the Bluetooth clock.

Return type
float

get_time_from_bytes(*time_bytes: bytes*) → float

Convert bytes read from the PVVX device to a timestamp.

Parameters
time_bytes (*bytes*) – The raw bytes read from the device.

Raises
InvalidTimeBytesError – If *time_bytes* don't have the right format.

Returns
The time encoded as a Unix timestamp.

Return type
float

classmethod recognize(*device: BLEDevice*, *advertisement_data: AdvertisementData*) → bool

Recognize the PVVX device from advertisement data.

This checks whether the advertisement has service data with service UUID 0x181a (PVVX custom format).

Parameters

- **device** (*BLEDevice*) – The Bluetooth device.
- **advertisement_data** (*AdvertisementData*) – The advertisement data.

Returns
`True` if the device is recognized as a PVVX device, `False` otherwise.

Return type
bool

bluetooth_clocks.devices.qingping module

Bluetooth clock support for Qingping clocks.

class bluetooth_clocks.devices.qingping.**CGC1**(*device: BLEDevice*)

Bases: *BluetoothClock*

Bluetooth clock support for the Qingping BT Clock Lite (CGC1).

CHAR_UUID: ClassVar[UUID] = UUID('00000001-0000-1000-8000-00805f9b34fb')

The UUID of the characteristic used to write the time.

DEVICE_TYPE: ClassVar[str] = 'Qingping BT Clock Lite'

The name of the device type.

LOCAL_NAME: ClassVar[str | None] = 'Qingping BT Clock Lite'

The local name used to recognize this type of device.

LOCAL_NAME_STARTS_WITH: ClassVar[bool | None] = False

The local name should exactly match *LOCAL_NAME*.

SERVICE_UUID: ClassVar[UUID] = UUID('22210000-554a-4546-5542-46534450464d')

The UUID of the service used to write the time.

TIME_GET_FORMAT: ClassVar[str | None] = None

The Qingping BT Clock Lite doesn't support reading the time.

TIME_SET_FORMAT: ClassVar[str] = '<BBL'

The format string to convert a time to bytes written to the device.

This starts with two bytes, followed by an unsigned long in little-endian format.

WRITE_WITH_RESPONSE: ClassVar[bool] = True

We use write with response to write the time to the Qingping BT Clock Lite.

Note: The device also supports write without response.

get_bytes_from_time(*timestamp: float*, *ampm: bool = False*) → bytes

Generate the bytes to set the time on the Qingping BT Clock Lite.

Parameters

- **timestamp** (*float*) – The time encoded as a Unix timestamp.
- **ampm** (*bool*) – `True` if the device should show the time with AM/PM, `False` if it should use 24-hour format. The Qingping BT Clock Lite ignores this argument, as it doesn`t support this option.

Returns

The bytes needed to set the time of the device to *timestamp*.

Return type

bytes

bluetooth_clocks.devices.thermopro module

Bluetooth clock support for ThermoPro sensors with clock.

class bluetooth_clocks.devices.thermopro.**TP358**(*device: BLEDevice*)

Bases: *TPXXX*

Bluetooth clock support for the ThermoPro TP358.

DEVICE_TYPE: ClassVar[str] = 'ThermoPro TP358'

The name of the device type.

LOCAL_NAME: ClassVar[str | None] = 'TP358'

The local name used to recognize this type of device.

LOCAL_NAME_STARTS_WITH: ClassVar[bool | None] = True

The local name should start with *LOCAL_NAME*.

class bluetooth_clocks.devices.thermopro.**TP393**(*device: BLEDevice*)

Bases: *TPXXX*

Bluetooth clock support for the ThermoPro TP393.

DEVICE_TYPE: ClassVar[str] = 'ThermoPro TP393'

The name of the device type.

LOCAL_NAME: ClassVar[str | None] = 'TP393'

The local name used to recognize this type of device.

LOCAL_NAME_STARTS_WITH: ClassVar[bool | None] = True

The local name should start with *LOCAL_NAME*.

class bluetooth_clocks.devices.thermopro.**TPXXX**(*device: BLEDevice*)

Bases: *BluetoothClock*

Bluetooth clock support for ThermoPro sensors with clock.

This class isn't meant to be instantiated. Subclasses implement support for specific ThermoPro device types by giving values to the class variables *DEVICE_TYPE*, *LOCAL_NAME*, and *LOCAL_NAME_STARTS_WITH*.

CHAR_UUID: ClassVar[UUID] = UUID('00010203-0405-0607-0809-0a0b0c0d2b11')

The UUID of the characteristic used to write the time.

SERVICE_UUID: ClassVar[UUID] = UUID('00010203-0405-0607-0809-0a0b0c0d1910')

The UUID of the service used to write the time.

TIME_GET_FORMAT: ClassVar[str | None] = None

ThermoPro devices don`t support reading the time.

TIME_SET_FORMAT: ClassVar[str] = 'BBBBBBBBBB'

The format string to convert a time to bytes written to the device.

These are ten bytes.

WRITE_WITH_RESPONSE: ClassVar[bool] = False

Writing the time to ThermoPro devices needs write without response.

get_bytes_from_time(*timestamp: float*, *ampm: bool = False*) → bytes

Generate the bytes to set the time on ThermoPro devices.

Parameters

- **timestamp** (*float*) – The time encoded as a Unix timestamp.
- **ampm** (*bool*) – `True` if the device should show the time with AM/PM, `False` if it should use 24-hour format.

Returns

The bytes needed to set the time of the device to *timestamp*.

Return type

bytes

bluetooth_clocks.devices.xiaomi module

Bluetooth clock support for Xiaomi devices.

class bluetooth_clocks.devices.xiaomi.**LYWSD02**(*device: BLEDevice*)

Bases: *BluetoothClock*

Bluetooth clock support for the Xiaomi LYWSD02.

CHAR_UUID: ClassVar[UUID] = UUID('ebe0ccb7-7a0a-4b0c-8a1a-6ff2997da3a6')

The UUID of the characteristic used to write the time.

DEVICE_TYPE: ClassVar[str] = 'Xiaomi LYWSD02'

The name of the device type.

LOCAL_NAME: ClassVar[str | None] = 'LYWSD02'

The local name used to recognize this type of device.

LOCAL_NAME_STARTS_WITH: ClassVar[bool | None] = False

The local name should exactly match *LOCAL_NAME*.

SERVICE_UUID: ClassVar[UUID] = UUID('ebe0ccb0-7a0a-4b0c-8a1a-6ff2997da3a6')

The UUID of the service used to write the time.

TIME_GET_FORMAT: ClassVar[str | None] = '<Lb'

The format string to convert bytes read from the device to a time.

TIME_SET_FORMAT: ClassVar[str] = '<Lb'

The format string to convert a time to bytes written to the device.

WRITE_WITH_RESPONSE: ClassVar[bool] = False

Writing the time to the device needs write without response.

get_bytes_from_time(*timestamp: float*, *ampm: bool = False*) → bytes

Generate the bytes to set the time on the Xiaomi LYWSD02.

Parameters

- **timestamp** (*float*) – The time encoded as a Unix timestamp.
- **ampm** (*bool*) – `True` if the device should show the time with AM/PM, `False` if it should use 24-hour format. The Xiaomi LYWSD02 ignores this argument, as it doesn't support this option.

Returns

The bytes needed to set the time of the device to *timestamp*.

Return type

bytes

get_time_from_bytes(*time_bytes: bytes*) → float

Convert bytes read from the Xiaomi LYWSD02 to a timestamp.

Parameters

time_bytes (*bytes*) – The raw bytes read from the device.

Raises

InvalidTimeBytesError – If *time_bytes* don't have the right format.

Returns

The time encoded as a Unix timestamp.

Return type
float

Submodules

bluetooth_clocks.exceptions module

Module with exceptions raised by this library.

exception bluetooth_clocks.exceptions.**BluetoothClocksError**

Bases: Exception

Base class for all exceptions raised by this library.

exception bluetooth_clocks.exceptions.**InvalidTimeBytesError**

Bases: *BluetoothClocksError*

Exception raised when bytes read from a device don't have the right format.

exception bluetooth_clocks.exceptions.**TimeNotReadableError**

Bases: *BluetoothClocksError*

Exception raised when reading the time on a device that doesn't support this.

exception bluetooth_clocks.exceptions.**UnsupportedDeviceError**

Bases: *BluetoothClocksError*

Exception raised when a device is not supported.

bluetooth_clocks.scanners module

Module with functions to scan for Bluetooth clocks.

async bluetooth_clocks.scanners.**discover_clocks**(*callback: Callable[[BluetoothClock], None]*, *scan_duration: float = 5.0*) → None

Discover Bluetooth clocks.

Parameters

- **callback** (*Callable[[BluetoothClock], None]*) – Function to call when a clock has been discovered. This function gets passed the discovered `BluetoothClock` object as its argument.
- **scan_duration** (*float*) – The scan duration for discovering devices. Defaults to 5 seconds.

async bluetooth_clocks.scanners.**find_clock**(*address: str*, *scan_duration: float = 5.0*) → *BluetoothClock* | None

Get BluetoothClock object from Bluetooth address.

Parameters

- **address** (*str*) – The Bluetooth address of the device.
- **scan_duration** (*float*) – The scan duration for finding the device. Defaults to 5 seconds.

Raises
UnsupportedDeviceError – If the device with address *address* isn`t supported.

Returns
A `BluetoothClock` object for the device, or `None` if the device isn't found.

Return type
BluetoothClock | None

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

b

INDEX

I

L

M

N

P

R

S

T

U

W