python-kasa

python-kasa developers

Apr 25, 2024

CONTENTS

python-kasa is a Python library to control TPLink's smart home devices (plugs, wall switches, power strips, and bulbs).

This is a voluntary, community-driven effort and is not affiliated, sponsored, or endorsed by TPLink.

Contributions in any form (adding missing features, reporting issues, fixing or triaging existing ones, improving the documentation, or device donations) are more than welcome!

You can install the most recent release using pip:

```
pip install python-kasa
```

For enhanced cli tool support (coloring, embedded shell) install with `[shell]`:

```
pip install python-kasa[shell]
```

If you are using cpython, it is recommended to install with `[speedups]` to enable orjson (faster json support):

```
pip install python-kasa[speedups]
```

or for both:

```
pip install python-kasa[speedups, shell]
```

With `[speedups]`, the protocol overhead is roughly an order of magnitude lower (benchmarks available in devtools).

Alternatively, you can clone this repository and use poetry to install the development version:

```
git clone https://github.com/python-kasa/python-kasa.git
cd python-kasa/
poetry install
```

If you have not yet provisioned your device, you can do so using the cli tool.

CHAPTER

ONE

DISCOVERING DEVICES

Running `kasa discover` will send discovery packets to the default broadcast address (`255.255.255.255`) to discover supported devices. If your system has multiple network interfaces, you can specify the broadcast address using the `--target` option.

The `discover` command will automatically execute the `state` command on all the discovered devices:

```
$ kasa discover
Discovering devices on 255.255.255.255 for 3 seconds

== Bulb McBulby - KL130(EU) ==
        Host: 192.168.xx.xx
        Port: 9999
        Device state: True
        == Generic information ==
        Time:         2023-12-05 14:33:23 (tz: {'index': 6, 'err_code': 0}
        Hardware:     1.0
        Software:     1.8.8 Build 190613 Rel.123436
        MAC (rssi):   1c:3b:f3:xx:xx:xx (-56)
        Location:     {'latitude': None, 'longitude': None}

        == Device specific information ==
        Brightness: 16
        Is dimmable: True
        Color temperature: 2500
        Valid temperature range: ColorTempRange(min=2500, max=9000)
        HSV: HSV(hue=0, saturation=0, value=16)
        Presets:
                index=0 brightness=50 hue=0 saturation=0 color_temp=2500 custom=None␣
↪id=None mode=None
                index=1 brightness=100 hue=299 saturation=95 color_temp=0 custom=None␣
↪id=None mode=None
                index=2 brightness=100 hue=120 saturation=75 color_temp=0 custom=None␣
↪id=None mode=None
                index=3 brightness=100 hue=240 saturation=75 color_temp=0 custom=None␣
↪id=None mode=None

        == Current State ==
        <EmeterStatus power=2.4 voltage=None current=None total=None>

        == Modules ==
        + <Module Schedule (smartlife.iot.common.schedule) for 192.168.xx.xx>
```

(continues on next page)

(continued from previous page)

```
        + <Module Usage (smartlife.iot.common.schedule) for 192.168.xx.xx>
        + <Module Antitheft (smartlife.iot.common.anti_theft) for 192.168.xx.xx>
        + <Module Time (smartlife.iot.common.timesetting) for 192.168.xx.xx>
        + <Module Emeter (smartlife.iot.common.emeter) for 192.168.xx.xx>
        - <Module Countdown (countdown) for 192.168.xx.xx>
        + <Module Cloud (smartlife.iot.common.cloud) for 192.168.xx.xx>
```

If your device requires authentication to control it, you need to pass the credentials using `--username` and `--password` options.

CHAPTER

TWO

BASIC FUNCTIONALITIES

All devices support a variety of common commands, including:

- `state` which returns state information
- `on` and `off` for turning the device on or off
- `emeter` (where applicable) to return energy consumption information
- `sysinfo` to return raw system information

The syntax to control device is `kasa --host <ip address> <command>`. Use `kasa --help` (or consult the documentation) to get a list of all available commands and options. Some examples of available options include JSON output (`--json`), defining timeouts (`--timeout` and `--discovery-timeout`).

Each individual command may also have additional options, which are shown when called with the `--help` option. For example, `--transition` on bulbs requests a smooth state change, while `--name` and `--index` are used on power strips to select the socket to act on:

```
$ kasa on --help

Usage: kasa on [OPTIONS]

  Turn the device on.

Options:
  --index INTEGER
  --name TEXT
  --transition INTEGER
  --help                Show this message and exit.
```

2.1 Bulbs

Common commands for bulbs and light strips include:

- `brightness` to control the brightness
- `hsv` to control the colors
- `temperature` to control the color temperatures

When executed without parameters, these commands will report the current state.

Some devices support `--transition` option to perform a smooth state change. For example, the following turns the light to 30% brightness over a period of five seconds:

```
$ kasa --host <addr> brightness --transition 5000 30
```

See `--help` for additional options and the documentation for more details about supported features and limitations.

2.2 Power strips

Each individual socket can be controlled separately by passing `--index` or `--name` to the command. If neither option is defined, the commands act on the whole power strip.

For example:

```
$ kasa --host <addr> off  # turns off all sockets
$ kasa --host <addr> off --name 'Socket1'  # turns off socket named 'Socket1'
```

See `--help` for additional options and the documentation for more details about supported features and limitations.

CHAPTER

THREE

ENERGY METER

Running `kasa emeter` command will return the current consumption. Possible options include `--year` and `--month` for retrieving historical state, and reseting the counters can be done with `--erase`.

```
$ kasa emeter
== Emeter ==
Current state: {'total': 133.105, 'power': 108.223577, 'current': 0.54463, 'voltage':␣
↪225.296283}
```

CHAPTER

FOUR

LIBRARY USAGE

If you want to use this library in your own project, a good starting point is to check the documentation on discovering devices. You can find several code examples in the API documentation of each of the implementation base classes, check out the documentation for the base class shared by all supported devices.

The library design and module structure is described in a separate page.

The device type specific documentation can be found in their separate pages:

- Plugs
- Bulbs
- Dimmers
- Power strips
- Light strips

4.1 Contributing

Contributions are very welcome! To simplify the process, we are leveraging automated checks and tests for contributions.

4.1.1 Setting up development environment

To get started, simply clone this repository and initialize the development environment. We are using poetry for dependency management, so after cloning the repository simply execute `poetry install` which will install all necessary packages and create a virtual environment for you.

4.1.2 Code-style checks

We use several tools to automatically check all contributions. The simplest way to verify that everything is formatted properly before creating a pull request, consider activating the pre-commit hooks by executing `pre-commit install`. This will make sure that the checks are passing when you do a commit.

You can also execute the checks by running either `tox -e lint` to only do the linting checks, or `tox` to also execute the tests.

4.1.3 Running tests

You can run tests on the library by executing `pytest` in the source directory. This will run the tests against contributed example responses, but you can also execute the tests against a real device:

```
$ pytest --ip <address>
```

Note that this will perform state changes on the device.

4.1.4 Analyzing network captures

The simplest way to add support for a new device or to improve existing ones is to capture traffic between the mobile app and the device. After capturing the traffic, you can either use the softScheck's wireshark dissector or the `parse_pcap.py` script contained inside the `devtools` directory. Note, that this works currently only on kasa-branded devices which use port 9999 for communications.

4.2 Supported devices

The following devices have been tested and confirmed as working. If your device is unlisted but working, please open a pull request to update the list and add a fixture file (use `python -m devtools.dump_devinfo` to generate one).

4.2.1 Supported Kasa devices

- **Plugs**: EP10, EP25*, HS100**, HS103, HS105, HS110, KP100, KP105, KP115, KP125, KP125M*, KP401
- **Power Strips**: EP40, HS107, HS300, KP200, KP303, KP400
- **Wall Switches**: ES20M, HS200, HS210, HS220, KP405, KS200M, KS205*, KS220M, KS225*, KS230, KS240*
- **Bulbs**: KL110, KL120, KL125, KL130, KL135, KL50, KL60, LB110
- **Light Strips**: KL400L5, KL420L5, KL430
- **Hubs**: KH100*

4.2.2 Supported Tapo* devices

- **Plugs**: P100, P110, P125M, P135, TP15
- **Power Strips**: P300, TP25
- **Wall Switches**: S500D, S505
- **Bulbs**: L510B, L510E, L530E
- **Light Strips**: L900-10, L900-5, L920-5, L930-5
- **Hubs**: H100

* Model requires authentication ** Newer versions require authentication

See *supported devices in our documentation* for more detailed information about tested hardware and software versions.

4.3 Resources

4.3.1 Developer Resources

- softScheck's github contains lot of information and wireshark dissector
- TP-Link Smart Home Device Simulator
- Unofficial API documentation
- Another unofficial API documentation
- pyHS100 provides synchronous interface and is the unmaintained predecessor of this library.

4.3.2 Library Users

- Home Assistant
- MQTT access to TP-Link devices, using python-kasa

4.3.3 TP-Link Tapo support

This library has recently added a limited supported for devices that carry Tapo branding. That support is currently limited to the cli. The package `kasa.smart` is in flux and if you use it directly you should expect it could break in future releases until this statement is removed.

Other TAPO libraries are:

- PyTapo - Python library for communication with Tapo Cameras
- Tapo P100 (Tapo plugs, Tapo bulbs)
  - Home Assistant integration
- plugp100, another tapo library
  - Home Assistant integration
- rust and python implementation

CHAPTER

FIVE

COMMAND-LINE USAGE

The package is shipped with a console tool named `kasa`, refer to `kasa --help` for detailed usage. The device to which the commands are sent is chosen by `KASA_HOST` environment variable or passing `--host <address>` as an option. To see what is being sent to and received from the device, specify option `--debug`.

To avoid discovering the devices when executing commands its type can be passed as an option (e.g., `--type plug` for plugs, `--type bulb` for bulbs, ..). If no type is manually given, its type will be discovered automatically which causes a short delay. Note that the `--type` parameter only works for legacy devices using port 9999.

To avoid discovering the devices for newer KASA or TAPO devices using port 20002 for discovery the `--device-family`, `-encrypt-type` and optional `-login-version` options can be passed and the devices will probably require authentication via `--username` and `--password`. Refer to `kasa --help` for detailed usage.

If no command is given, the `state` command will be executed to query the device state.

Note: Some commands (such as reading energy meter values, changing bulb settings, or accessing individual sockets on smart strips) additional parameters are required, which you can find by adding `--help` after the command, e.g. `kasa --type emeter --help` or `kasa --type hsv --help`. Refer to the device type specific documentation for more details.

5.1 Discovery

The tool can automatically discover supported devices using a broadcast-based discovery protocol. This works by sending an UDP datagram on ports 9999 and 20002 to the broadcast address (defaulting to `255.255.255.255`).

Newer devices that respond on port 20002 will require TP-Link cloud credentials to be passed (unless they have never been connected to the TP-Link cloud) or they will report as having failed authentication when trying to query the device. Use `--username` and `--password` options to specify credentials. These values can also be set as environment variables via `KASA_USERNAME` and `KASA_PASSWORD`.

On multihomed systems, you can use `--target` option to specify the broadcast target. For example, if your devices reside in network `10.0.0.0/24` you can use `kasa --target 10.0.0.255 discover` to discover them.

Note: When no command is specified when invoking `kasa`, a discovery is performed and the `state` command is executed on each discovered device.

5.2 Provisioning

You can provision your device without any extra apps by using the `kasa wifi` command:

1. If the device is unprovisioned, connect to its open network
2. Use `kasa discover` (or check the routes) to locate the IP address of the device (likely 192.168.0.1, if unprovisioned)
3. Scan for available networks using `kasa --host 192.168.0.1 wifi scan` see which networks are visible to the device
4. Join/change the network using `kasa --host 192.168.0.1 wifi join <network to join>`

As with all other commands, you can also pass `--help` to both `join` and `scan` commands to see the available options.

Note: For devices requiring authentication, the device-stored credentials can be changed using the `update-credentials` commands, for example, to match with other cloud-connected devices. However, note that communications with devices provisioned using this method will stop working when connected to the cloud.

5.3 kasa --help

```
Usage: kasa [OPTIONS] COMMAND [ARGS]...

  A tool for controlling TP-Link smart home devices.

Options:
  --host TEXT                     The host name or IP address of the device to
                                  connect to.
  --port INTEGER                  The port of the device to connect to.
  --alias TEXT                    The device name, or alias, of the device to
                                  connect to.
  --target TEXT                   The broadcast address to be used for
                                  discovery.  [default: 255.255.255.255]
  -v, --verbose                   Be more verbose on output
  -d, --debug                     Print debug output
  --type [plug|switch|bulb|dimmer|strip|lightstrip|iot.plug|iot.switch|iot.bulb|iot.
↪dimmer|iot.strip|iot.lightstrip|smart.plug|smart.bulb]
  --json / --no-json              Output raw device response as JSON.
  --encrypt-type [KLAP|AES|XOR]
  --device-family [IOT.SMARTPLUGSWITCH|IOT.SMARTBULB|SMART.KASAPLUG|SMART.
↪KASASWITCH|SMART.TAPOPLUG|SMART.TAPOBULB|SMART.TAPOSWITCH|SMART.TAPOHUB|SMART.KASAHUB]
  --login-version INTEGER
  --timeout INTEGER               Timeout for device communications.
                                  [default: 5]
  --discovery-timeout INTEGER     Timeout for discovery.  [default: 5]
  --username TEXT                 Username/email address to authenticate to
                                  device.
  --password TEXT                 Password to use to authenticate to device.
  --credentials-hash TEXT         Hashed credentials used to authenticate to
                                  the device.
  --version                       Show the version and exit.
```

(continues on next page)

(continued from previous page)

```
  --help                          Show this message and exit.

Commands:
  alias                Get or set the device (or plug) alias.
  brightness           Get or set brightness.
  command              Run a raw command on the device.
  discover             Discover devices in the network.
  effect               Set an effect.
  emeter               Query emeter for historical consumption.
  feature              Access and modify features.
  hsv                  Get or set color in HSV.
  led                  Get or set (Plug's) led state.
  off                  Turn the device off.
  on                   Turn the device on.
  presets              List and modify bulb setting presets.
  raw-command          Run a raw command on the device.
  reboot               Reboot the device.
  schedule             Scheduling commands.
  shell                Open interactive shell.
  state                Print out device state and versions.
  sysinfo              Print out full system information.
  temperature          Get or set color temperature.
  time                 Get the device time.
  toggle               Toggle the device on/off.
  turn-on-behavior     Modify bulb turn-on behavior.
  update-credentials   Update device credentials for authenticated devices.
  usage                Query usage for historical consumption.
  wifi                 Commands to control wifi settings.
Raised error: 0
Run with --debug enabled to see stacktrace
```

CHAPTER

SIX

DISCOVERING DEVICES

Contents

6.1 Discovery

Discovery works by sending broadcast UDP packets to two known TP-link discovery ports, 9999 and 20002. Port 9999 is used for legacy devices that do not use strong encryption and 20002 is for newer devices that use different levels of encryption. If a device uses port 20002 for discovery you will obtain some basic information from the device via discovery, but you will need to await `SmartDevice.update()` to get full device information. Credentials will most likely be required for port 20002 devices although if the device has never been connected to the tplink cloud it may work without credentials.

To query or update the device requires authentication via *Credentials* and if this is invalid or not provided it will raise an `AuthenticationException`.

If discovery encounters an unsupported device when calling via *Discover.discover_single()* it will raise a `UnsupportedDeviceException`. If discovery encounters a device when calling *Discover.discover()*, you can provide a callback to the `on_unsupported` parameter to handle these.

Example:

```
import asyncio
from kasa import Discover, Credentials

async def main():
    device = await Discover.discover_single(
        "127.0.0.1",
        credentials=Credentials("myusername", "mypassword"),
        discovery_timeout=10
    )

    await device.update()  # Request the update
    print(device.alias)  # Print out the alias

    devices = await Discover.discover(
        credentials=Credentials("myusername", "mypassword"),
```

(continues on next page)

(continued from previous page)

```
        discovery_timeout=10
    )
    for ip, device in devices.items():
        await device.update()
        print(device.alias)

if __name__ == "__main__":
    asyncio.run(main())
```

6.2 API documentation

class kasa.**Discover**

Discover TPLink Smart Home devices.

The main entry point for this library is *Discover.discover()*, which returns a dictionary of the found devices. The key is the IP address of the device and the value contains ready-to-use, SmartDevice-derived device object.

discover_single() can be used to initialize a single device given its IP address. If the *DeviceConfig* of the device is already known, you can initialize the corresponding device class directly without discovery.

The protocol uses UDP broadcast datagrams on port 9999 and 20002 for discovery. Legacy devices support discovery on port 9999 and newer devices on 20002.

Newer devices that respond on port 20002 will most likely require TP-Link cloud credentials to be passed if queries or updates are to be performed on the returned devices.

Examples:

Discovery returns a list of discovered devices:

```
>>> import asyncio
>>> found_devices = asyncio.run(Discover.discover())
>>> [dev.alias for dev in found_devices]
['TP-LINK_Power Strip_CF69']
```

Discovery can also be targeted to a specific broadcast address instead of the default 255.255.255.255:

```
>>> asyncio.run(Discover.discover(target="192.168.8.255"))
```

It is also possible to pass a coroutine to be executed for each found device:

```
>>> async def print_alias(dev):
>>>     print(f"Discovered {dev.alias}")
>>> devices = asyncio.run(Discover.discover(on_discovered=print_alias))
```

DISCOVERY_PORT = 9999

DISCOVERY_PORT_2 = 20002

DISCOVERY_QUERY = {'system': {'get_sysinfo': None}}

DISCOVERY_QUERY_2 = b'\x02\x00\x00\x01\x00\x00\x00\x00\x00\x00\x00\x00F<\xb5\xd3'

async static discover(*, target='255.255.255.255', on_discovered=None, discovery_timeout=5, discovery_packets=3, interface=None, on_unsupported=None, credentials=None, port=None, timeout=None) → Dict[str, Device]

Discover supported devices.

Sends discovery message to 255.255.255.255:9999 and 255.255.255.255:20002 in order to detect available supported devices in the local network, and waits for given timeout for answers from devices. If you have multiple interfaces, you can use *target* parameter to specify the network for discovery.

If given, *on_discovered* coroutine will get awaited with a SmartDevice-derived object as parameter.

The results of the discovery are returned as a dict of SmartDevice-derived objects keyed with IP addresses. The devices are already initialized and all but emeter-related properties can be accessed directly.

Parameters

- **target** – The target address where to send the broadcast discovery queries if multi-homing (e.g. 192.168.xxx.255).
- **on_discovered** – coroutine to execute on discovery
- **discovery_timeout** – Seconds to wait for responses, defaults to 5
- **discovery_packets** – Number of discovery packets to broadcast
- **interface** – Bind to specific interface
- **on_unsupported** – Optional callback when unsupported devices are discovered
- **credentials** – Credentials for devices requiring authentication
- **port** – Override the discovery port for devices listening on 9999
- **timeout** – Query timeout in seconds for devices returned by discovery

Returns
dictionary with discovered devices

async static discover_single(*host: str*, *, *discovery_timeout: int = 5*, *port: int | None = None*, *timeout: int | None = None*, *credentials:* kasa.credentials.Credentials *| None = None*) → Device

Discover a single device by the given IP address.

It is generally preferred to avoid discover_single() and use SmartDevice.connect() instead as it should perform better when the WiFi network is congested or the device is not responding to discovery requests.

Parameters

- **host** – Hostname of device to query
- **discovery_timeout** – Timeout in seconds for discovery
- **port** – Optionally set a different port for legacy devices using port 9999
- **timeout** – Timeout in seconds device for devices queries
- **credentials** – Credentials for devices that require authentication

Return type
SmartDevice

Returns
Object for querying/controlling found device.

CHAPTER

SEVEN

COMMON API

Contents

7.1 SmartDevice class

The basic functionalities of all supported devices are accessible using the common `SmartDevice` base class.

The property accesses use the data obtained before by awaiting `SmartDevice.update()`. The values are cached until the next update call. In practice this means that property accesses do no I/O and are dependent, while I/O producing methods need to be awaited. See *Library Design & Modules* for more detailed information.

Note: The device instances share the communication socket in background to optimize I/O accesses. This means that you need to use the same event loop for subsequent requests. The library gives a warning ("Detected protocol reuse between different event loop") to hint if you are accessing the device incorrectly.

Methods changing the state of the device do not invalidate the cache (i.e., there is no implicit `SmartDevice.update()` call made by the library). You can assume that the operation has succeeded if no exception is raised. These methods will return the device response, which can be useful for some use cases.

Errors are raised as `KasaException` instances for the library user to handle.

Simple example script showing some functionality for legacy devices:

```
import asyncio
from kasa import SmartPlug

async def main():
    p = SmartPlug("127.0.0.1")
```

(continues on next page)

(continued from previous page)

```
    await p.update()  # Request the update
    print(p.alias)  # Print out the alias
    print(p.emeter_realtime)  # Print out current emeter status

    await p.turn_off()  # Turn the device off

if __name__ == "__main__":
    asyncio.run(main())
```

If you are connecting to a newer KASA or TAPO device you can get the device via discovery or connect directly with *DeviceConfig*:

```
import asyncio
from kasa import Discover, Credentials

async def main():
    device = await Discover.discover_single(
        "127.0.0.1",
        credentials=Credentials("myusername", "mypassword"),
        discovery_timeout=10
    )

    config = device.config # DeviceConfig.to_dict() can be used to store for later

    # To connect directly later without discovery

    later_device = await SmartDevice.connect(config=config)

    await later_device.update()

    print(later_device.alias)  # Print out the alias
```

If you want to perform updates in a loop, you need to make sure that the device accesses are done in the same event loop:

```
import asyncio
from kasa import SmartPlug

async def main():
    dev = SmartPlug("127.0.0.1")  # We create the instance inside the main loop
    while True:
        await dev.update()  # Request an update
        print(dev.emeter_realtime)
        await asyncio.sleep(0.5)  # Sleep some time between updates

if __name__ == "__main__":
    asyncio.run(main())
```

Refer to device type specific classes for more examples: *SmartPlug*, *SmartBulb*, *SmartStrip*, *SmartDimmer*, *SmartLightStrip*.

7.2 DeviceConfig class

The *DeviceConfig* class can be used to initialise devices with parameters to allow them to be connected to without using discovery. This is required for newer KASA and TAPO devices that use different protocols for communication and will not respond on port 9999 but instead use different encryption protocols over http port 80. Currently there are three known types of encryption for TP-Link devices and two different protocols. Devices with automatic firmware updates enabled may update to newer versions of the encryption without separate notice, so discovery can be helpful to determine the correct config.

To connect directly pass a *DeviceConfig* object to `SmartDevice.connect()`.

A *DeviceConfig* can be constucted manually if you know the *DeviceConfig.connection_type* values for the device or alternatively the config can be retrieved from `SmartDevice.config` post discovery and then re-used.

7.3 Energy Consumption and Usage Statistics

Note: In order to use the helper methods to calculate the statistics correctly, your devices need to have correct time set. The devices use NTP and public servers from NTP Pool Project to synchronize their time.

7.3.1 Energy Consumption

The availability of energy consumption sensors depend on the device. While most of the bulbs support it, only specific switches (e.g., HS110) or strips (e.g., HS300) support it. You can use `has_emeter` to check for the availability.

7.3.2 Usage statistics

You can use `on_since` to query for the time the device has been turned on. Some devices also support reporting the usage statistics on daily or monthly basis. You can access this information using through the usage module (`kasa.modules.Usage`):

```
dev = SmartPlug("127.0.0.1")
usage = dev.modules["usage"]
print(f"Minutes on this month: {usage.usage_this_month}")
print(f"Minutes on today: {usage.usage_today}")
```

7.4 API documentation

kasa.**SmartDevice**

alias of `IotDevice`

class kasa.**DeviceConfig**(*host: str*, *timeout: ~typing.Optional[int] = 5*, *port_override: ~typing.Optional[int] = None*, *credentials: ~typing.Optional[~kasa.credentials.Credentials] = None*, *credentials_hash: ~typing.Optional[str] = None*, *batch_size: ~typing.Optional[int] = None*, *connection_type: ~kasa.deviceconfig.ConnectionType = <factory>*, *uses_http: bool = False*, *http_client: ~typing.Optional[ClientSession] = None*)

Class to represent paramaters that determine how to connect to devices.

DEFAULT_TIMEOUT = 5

host: str

IP address or hostname

timeout: Optional[int] = 5

Timeout for querying the device

port_override: Optional[int] = None

Override the default 9999 port to support port forwarding

credentials: Optional[*Credentials*] = None

Credentials for devices requiring authentication

credentials_hash: Optional[str] = None

Credentials hash for devices requiring authentication. If credentials are also supplied they take precendence over credentials_hash. Credentials hash can be retrieved from `SmartDevice.credentials_hash`

batch_size: Optional[int] = None

The protocol specific type of connection. Defaults to the legacy type.

connection_type: ConnectionType

The batch size for protoools supporting multiple request batches.

uses_http: bool = False

True if the device uses http. Consumers should retrieve rather than set this in order to determine whether they should pass a custom http client if desired.

http_client: Optional[ClientSession] = None

Set a custom http_client for the device to use.

to_dict(*, *credentials_hash: Optional[str] = None*, *exclude_credentials: bool = False*) → Dict[str, Dict[str, str]]

Convert device config to dict.

static from_dict(*config_dict: Dict[str, Dict[str, str]]*) → *DeviceConfig*

Return device config from dict.

class kasa.**Credentials**(*username: str = ''*, *password: str = ''*)

Credentials for authentication.

password: str = ''

Password of the cloud account

username: str = ''

Username (email address) of the cloud account

CHAPTER

EIGHT

LIBRARY DESIGN & MODULES

This page aims to provide some details on the design and internals of this library. You might be interested in this if you want to improve this library, or if you are just looking to access some information that is not currently exposed.

Contents

8.1 Initialization

Use *discover()* to perform udp-based broadcast discovery on the network. This will return you a list of device instances based on the discovery replies.

If the device's host is already known, you can use to construct a device instance with `connect()`.

The `connect()` also enables support for connecting to new KASA SMART protocol and TAPO devices directly using the parameter *DeviceConfig*. Simply serialize the `config` property via *to_dict()* and then deserialize it later with *from_dict()* and then pass it into `connect()`.

8.2 Update Cycle

When `update()` is called, the library constructs a query to send to the device based on *supported modules*. Internally, each module defines `query()` to describe what they want query during the update.

The returned data is cached internally to avoid I/O on property accesses. All properties defined both in the device class and in the module classes follow this principle.

While the properties are designed to provide a nice API to use for common use cases, you may sometimes want to access the raw, cached data as returned by the device. This can be done using the `internal_state` property.

8.3 Modules

The functionality provided by all *SmartDevice* instances is (mostly) done inside separate modules. While the individual device-type specific classes provide an easy access for the most import features, you can also access individual modules through `kasa.SmartDevice.modules`. You can get the list of supported modules for a given device instance using `supported_modules`.

Note: If you only need some module-specific information, you can call the wanted method on the module to avoid using `update()`.

8.4 Protocols and Transports

The library supports two different TP-Link protocols, `IOT` and `SMART`. `IOT` is the original Kasa protocol and `SMART` is the newer protocol supported by TAPO devices and newer KASA devices. The original protocol has a `target`, `command`, `args` interface whereas the new protocol uses a different set of commands and has a `method`, `parameters` interface. Confusingly TP-Link originally called the Kasa line "Kasa Smart" and hence this library used "Smart" in a lot of the module and class names but actually they were built to work with the `IOT` protocol.

In 2021 TP-Link started updating the underlying communication transport used by Kasa devices to make them more secure. It switched from a TCP connection with static XOR type of encryption to a transport called `KLAP` which communicates over http and uses handshakes to negotiate a dynamic encryption cipher. This automatic update was put on hold and only seemed to affect UK HS100 models.

In 2023 TP-Link started updating the underlying communication transport used by Tapo devices to make them more secure. It switched from AES encryption via public key exchange to use `KLAP` encryption and negotiation due to concerns around impersonation with AES. The encryption cipher is the same as for Kasa KLAP but the handshake seeds are slightly different. Also in 2023 TP-Link started releasing newer Kasa branded devices using the `SMART` protocol. This appears to be driven by hardware version rather than firmware.

In order to support these different configurations the library migrated from a single protocol class `TPLinkSmartHomeProtocol` to support pluggable transports and protocols. The classes providing this functionality are:

- *BaseProtocol*
- *IotProtocol*
- *SmartProtocol*
- *BaseTransport*
- *XorTransport*

- *AesTransport*
- *KlapTransport*
- *KlapTransportV2*

8.5 Errors and Exceptions

The base exception for all library errors is *KasaException*.

- If the device returns an error the library raises a *DeviceError* which will usually contain an `error_code` with the detail.
- If the device fails to authenticate the library raises an *AuthenticationError* which is derived from *DeviceError* and could contain an `error_code` depending on the type of failure.
- If the library encounters and unsupported deviceit raises an *UnsupportedDeviceError*.
- If the device fails to respond within a timeout the library raises a *TimeoutError*.
- All other failures will raise the base *KasaException* class.

8.6 API documentation for modules

8.7 API documentation for protocols and transports

class kasa.protocol.**BaseProtocol**(*, *transport:* BaseTransport)

Base class for all TP-Link Smart Home communication.

abstract async close() → None

Close the protocol. Abstract method to be overriden.

property config: *DeviceConfig*

Return the connection parameters the device is using.

abstract async query(*request: str | dict*, *retry_count: int = 3*) → dict

Query the device for the protocol. Abstract method to be overriden.

class kasa.iotprotocol.**IotProtocol**(*, *transport:* BaseTransport)

Class for the legacy TPLink IOT KASA Protocol.

BACKOFF_SECONDS_AFTER_TIMEOUT = 1

async close() → None

Close the underlying transport.

property config: *DeviceConfig*

Return the connection parameters the device is using.

async query(*request: str | dict*, *retry_count: int = 3*) → dict

Query the device retrying for retry_count on failure.

class kasa.smartprotocol.**SmartProtocol**(*, *transport:* BaseTransport)

Class for the new TPLink SMART protocol.

BACKOFF_SECONDS_AFTER_TIMEOUT = 1

DEFAULT_MULTI_REQUEST_BATCH_SIZE = 5

async close() → None
Close the underlying transport.

property config: *DeviceConfig*
Return the connection parameters the device is using.

get_smart_request(*method*, *params=None*) → str
Get a request message as a string.

async query(*request: str | dict*, *retry_count: int = 3*) → dict
Query the device retrying for retry_count on failure.

class kasa.protocol.**BaseTransport**(*, *config:* DeviceConfig)
Base class for all TP-Link protocol transports.

DEFAULT_TIMEOUT = 5

abstract async close() → None
Close the transport. Abstract method to be overriden.

abstract property credentials_hash: str
The hashed credentials used by the transport.

abstract property default_port: int
The default port for the transport.

abstract async reset() → None
Reset internal state.

abstract async send(*request: str*) → dict
Send a message to the device and return a response.

class kasa.xortransport.**XorTransport**(*, *config:* DeviceConfig)
XorTransport class.

BLOCK_SIZE = 4

DEFAULT_PORT: int = 9999

DEFAULT_TIMEOUT = 5

async close() → None
Close the connection.

close_without_wait() → None
Close the connection without waiting for the connection to close.

property credentials_hash: str
The hashed credentials used by the transport.

property default_port
Default port for the transport.

async reset() → None

Reset the transport.

The transport cannot be reset so we must close instead.

async send(*request: str*) → dict

Send a message to the device and return a response.

class kasa.klaptransport.**KlapTransport**(*, *config:* DeviceConfig)

Implementation of the KLAP encryption protocol.

KLAP is the name used in device discovery for TP-Link's new encryption protocol, used by newer firmware versions.

DEFAULT_PORT: int = 80

DEFAULT_TIMEOUT = 5

DISCOVERY_QUERY = {'system': {'get_sysinfo': None}}

SESSION_COOKIE_NAME = 'TP_SESSIONID'

TIMEOUT_COOKIE_NAME = 'TIMEOUT'

async close() → None

Close the http client and reset internal state.

property credentials_hash: str

The hashed credentials used by the transport.

property default_port

Default port for the transport.

static generate_auth_hash(*creds:* Credentials)

Generate an md5 auth hash for the protocol on the supplied credentials.

static generate_owner_hash(*creds:* Credentials)

Return the MD5 hash of the username in this object.

static handshake1_seed_auth_hash(*local_seed: bytes*, *remote_seed: bytes*, *auth_hash: bytes*)

Generate an md5 auth hash for the protocol on the supplied credentials.

static handshake2_seed_auth_hash(*local_seed: bytes*, *remote_seed: bytes*, *auth_hash: bytes*)

Generate an md5 auth hash for the protocol on the supplied credentials.

async perform_handshake() → Any

Perform handshake1 and handshake2.

Sets the encryption_session if successful.

async perform_handshake1() → tuple[bytes, bytes, bytes]

Perform handshake1.

async perform_handshake2(*local_seed*, *remote_seed*, *auth_hash*) → KlapEncryptionSession

Perform handshake2.

async reset() → None

Reset internal handshake state.

async send(*request: str*)

Send the request.

class kasa.klaptransport.KlapTransportV2(*, *config:* DeviceConfig)

Implementation of the KLAP encryption protocol with v2 hanshake hashes.

DEFAULT_PORT: int = 80

DEFAULT_TIMEOUT = 5

DISCOVERY_QUERY = {'system': {'get_sysinfo': None}}

SESSION_COOKIE_NAME = 'TP_SESSIONID'

TIMEOUT_COOKIE_NAME = 'TIMEOUT'

async close() → None

Close the http client and reset internal state.

property credentials_hash: str

The hashed credentials used by the transport.

property default_port

Default port for the transport.

static generate_auth_hash(*creds:* Credentials)

Generate an md5 auth hash for the protocol on the supplied credentials.

static generate_owner_hash(*creds:* Credentials)

Return the MD5 hash of the username in this object.

static handshake1_seed_auth_hash(*local_seed: bytes*, *remote_seed: bytes*, *auth_hash: bytes*)

Generate an md5 auth hash for the protocol on the supplied credentials.

static handshake2_seed_auth_hash(*local_seed: bytes*, *remote_seed: bytes*, *auth_hash: bytes*)

Generate an md5 auth hash for the protocol on the supplied credentials.

async perform_handshake() → Any

Perform handshake1 and handshake2.

Sets the encryption_session if successful.

async perform_handshake1() → tuple[bytes, bytes, bytes]

Perform handshake1.

async perform_handshake2(*local_seed*, *remote_seed*, *auth_hash*) → KlapEncryptionSession

Perform handshake2.

async reset() → None

Reset internal handshake state.

async send(*request: str*)

Send the request.

class kasa.aestransport.AesTransport(*, *config:* DeviceConfig)

Implementation of the AES encryption protocol.

AES is the name used in device discovery for TP-Link's TAPO encryption protocol, sometimes used by newer firmware versions on kasa devices.

BACKOFF_SECONDS_AFTER_LOGIN_ERROR = 1

COMMON_HEADERS = {'Accept': 'application/json', 'Content-Type': 'application/json', 'requestByApp': 'true'}

CONTENT_LENGTH = 'Content-Length'

DEFAULT_PORT: int = 80

DEFAULT_TIMEOUT = 5

KEY_PAIR_CONTENT_LENGTH = 314

SESSION_COOKIE_NAME = 'TP_SESSIONID'

TIMEOUT_COOKIE_NAME = 'TIMEOUT'

async close() → None
Close the http client and reset internal state.

property credentials_hash: str
The hashed credentials used by the transport.

property default_port: int
Default port for the transport.

static hash_credentials(*login_v2: bool*, *credentials:* Credentials) → tuple[str, str]
Hash the credentials.

async perform_handshake() → None
Perform the handshake.

async perform_login()
Login to the device.

async reset() → None
Reset internal handshake and login state.

async send(*request: str*) → dict[str, Any]
Send the request.

async send_secure_passthrough(*request: str*) → dict[str, Any]
Send encrypted message as passthrough.

async try_login(*login_params: dict[str, Any]*) → None
Try to login with supplied login_params.

8.8 API documentation for errors and exceptions

class kasa.exceptions.**KasaException**
Base exception for library errors.

class kasa.exceptions.**DeviceError**(**args: Any*, ***kwargs: Any*)
Base exception for device errors.

class kasa.exceptions.**AuthenticationError**(**args: Any*, ***kwargs: Any*)

Base exception for device authentication errors.

class kasa.exceptions.**UnsupportedDeviceError**(**args: Any*, ***kwargs: Any*)

Exception for trying to connect to unsupported devices.

class kasa.exceptions.**TimeoutError**

Timeout exception for device errors.

CHAPTER

NINE

BULBS

Contents

9.1 Supported features

- Turning on and off
- Setting brightness, color temperature, and color (in HSV)
- Querying emeter information
- Transitions
- Presets

9.2 Currently unsupported

- Setting the default transitions
- Timers

Note: Feel free to open a pull request to add support for more features!

9.3 Transitions

All commands changing the bulb state can be accompanied with a transition, e.g., to slowly fade the light off. The transition time is in milliseconds, 0 means immediate change. If no transition value is given, the default setting as configured for the bulb will be used.

Note: Accepted values are command (and potentially bulb) specific, feel free to improve the documentation on accepted values.

Example: While KL130 allows at least up to 15 second transitions for smooth turning off transitions, turning it on will not be so smooth.

9.4 Command-line usage

All command-line commands can be used with transition period for smooth changes.

Example: Turn the bulb off over a 15 second time period.

```
$ kasa --type bulb --host <host> off --transition 15000
```

Example: Change the bulb to red with 20% brightness over 15 seconds:

```
$ kasa --type bulb --host <host> hsv 0 100 20 --transition 15000
```

9.5 API documentation

kasa.**SmartBulb**

alias of IotBulb

kasa.**SmartBulbPreset**

alias of BulbPreset

class kasa.**TurnOnBehaviors**(*, *soft_on:* TurnOnBehavior, *hard_on:* TurnOnBehavior)

Model to contain turn on behaviors.

hard: *TurnOnBehavior*

The behavior when the bulb has been off from mains power.

soft: *TurnOnBehavior*

The behavior when the bulb is turned on programmatically.

class kasa.**TurnOnBehavior**(*, *index: Optional[int] = None*, *mode: BehaviorMode*)

Model to present a single turn on behavior.

Parameters

- **preset** (*int*) – the index number of wanted preset.
- **mode** (*BehaviorMode*) – last status or preset mode. If you are changing existing settings, you should not set this manually.

To change the behavior, it is only necessary to change the *preset* field to contain either the preset index, or None for the last known state.

class Config

Configuration to make the validator run when changing the values.

validate_assignment = True

mode: BehaviorMode

Wanted behavior

preset: Optional[int]

Index of preset to use, or None for the last known state.

CHAPTER

TEN

PLUGS

Contents

Note: Feel free to open a pull request to improve the documentation!

10.1 API documentation

kasa.**SmartPlug**

alias of `IotPlug`

CHAPTER

ELEVEN

DIMMERS

Contents

Note: Feel free to open a pull request to improve the documentation!

11.1 API documentation

kasa.**SmartDimmer**

alias of IotDimmer

CHAPTER

TWELVE

SMART STRIPS

Contents

Note: Feel free to open a pull request to improve the documentation!

12.1 Command-line usage

To command a single socket of a strip, you will need to specify it either by using `--index` or by using `--name`. If not specified, the commands will act on the parent device: turning the strip off will turn off all sockets.

Example: Turn on the first socket (the indexing starts from zero):

```
$ kasa --type strip --host <host> on --index 0
```

Example: Turn off the socket by name:

```
$ kasa --type strip --host <host> off --name "Maybe Kitchen"
```

12.2 API documentation

kasa.**SmartStrip**

alias of `IotStrip`

CHAPTER

THIRTEEN

LIGHT STRIPS

Contents

Note: Feel free to open a pull request to improve the documentation!

13.1 API documentation

kasa.**SmartLightStrip**

alias of `IotLightStrip`

CHAPTER

FOURTEEN

SUPPORTED DEVICES

The following devices have been tested and confirmed as working. If your device is unlisted but working, please open a pull request to update the list and add a fixture file (use `python -m devtools.dump_devinfo` to generate one).

14.1 Kasa devices

Some newer Kasa devices require authentication. These are marked with * in the list below.

14.1.1 Plugs

- **EP10**
  - Hardware: 1.0 (US) / Firmware: 1.0.2
- **EP25**
  - Hardware: 2.6 (US) / Firmware: 1.0.1*
  - Hardware: 2.6 (US) / Firmware: 1.0.2*
- **HS100**
  - Hardware: 1.0 (UK) / Firmware: 1.2.6
  - Hardware: 4.1 (UK) / Firmware: 1.1.0*
  - Hardware: 1.0 (US) / Firmware: 1.2.5
  - Hardware: 2.0 (US) / Firmware: 1.5.6
- **HS103**
  - Hardware: 1.0 (US) / Firmware: 1.5.7
  - Hardware: 2.1 (US) / Firmware: 1.1.2
  - Hardware: 2.1 (US) / Firmware: 1.1.4
- **HS105**
  - Hardware: 1.0 (US) / Firmware: 1.5.6
- **HS110**
  - Hardware: 1.0 (EU) / Firmware: 1.2.5
  - Hardware: 4.0 (EU) / Firmware: 1.0.4
- **KP100**

  - Hardware: 3.0 (US) / Firmware: 1.0.1
- **KP105**
  - Hardware: 1.0 (UK) / Firmware: 1.0.5
  - Hardware: 1.0 (UK) / Firmware: 1.0.7
- **KP115**
  - Hardware: 1.0 (EU) / Firmware: 1.0.16
  - Hardware: 1.0 (US) / Firmware: 1.0.17
  - Hardware: 1.0 (US) / Firmware: 1.0.21
- **KP125**
  - Hardware: 1.0 (US) / Firmware: 1.0.6
- **KP125M**
  - Hardware: 1.0 (US) / Firmware: 1.1.3*
- **KP401**
  - Hardware: 1.0 (US) / Firmware: 1.0.0

14.1.2 Power Strips

- **EP40**
  - Hardware: 1.0 (US) / Firmware: 1.0.2
- **HS107**
  - Hardware: 1.0 (US) / Firmware: 1.0.8
- **HS300**
  - Hardware: 1.0 (US) / Firmware: 1.0.10
  - Hardware: 2.0 (US) / Firmware: 1.0.12
  - Hardware: 2.0 (US) / Firmware: 1.0.3
- **KP200**
  - Hardware: 3.0 (US) / Firmware: 1.0.3
- **KP303**
  - Hardware: 1.0 (UK) / Firmware: 1.0.3
  - Hardware: 2.0 (US) / Firmware: 1.0.3
- **KP400**
  - Hardware: 1.0 (US) / Firmware: 1.0.10
  - Hardware: 2.0 (US) / Firmware: 1.0.6

14.1.3 Wall Switches

- **ES20M**
  - Hardware: 1.0 (US) / Firmware: 1.0.8
- **HS200**
  - Hardware: 2.0 (US) / Firmware: 1.5.7
  - Hardware: 5.0 (US) / Firmware: 1.0.2
- **HS210**
  - Hardware: 1.0 (US) / Firmware: 1.5.8
- **HS220**
  - Hardware: 1.0 (US) / Firmware: 1.5.7
  - Hardware: 2.0 (US) / Firmware: 1.0.3
- **KP405**
  - Hardware: 1.0 (US) / Firmware: 1.0.5
- **KS200M**
  - Hardware: 1.0 (US) / Firmware: 1.0.8
- **KS205**
  - Hardware: 1.0 (US) / Firmware: 1.0.2*
- **KS220M**
  - Hardware: 1.0 (US) / Firmware: 1.0.4
- **KS225**
  - Hardware: 1.0 (US) / Firmware: 1.0.2*
- **KS230**
  - Hardware: 1.0 (US) / Firmware: 1.0.14
- **KS240**
  - Hardware: 1.0 (US) / Firmware: 1.0.4*
  - Hardware: 1.0 (US) / Firmware: 1.0.5*

14.1.4 Bulbs

- **KL110**
  - Hardware: 1.0 (US) / Firmware: 1.8.11
- **KL120**
  - Hardware: 1.0 (US) / Firmware: 1.8.6
- **KL125**
  - Hardware: 1.20 (US) / Firmware: 1.0.5
  - Hardware: 2.0 (US) / Firmware: 1.0.7

  - Hardware: 4.0 (US) / Firmware: 1.0.5
- **KL130**
  - Hardware: 1.0 (EU) / Firmware: 1.8.8
  - Hardware: 1.0 (US) / Firmware: 1.8.11
- **KL135**
  - Hardware: 1.0 (US) / Firmware: 1.0.6
- **KL50**
  - Hardware: 1.0 (US) / Firmware: 1.1.13
- **KL60**
  - Hardware: 1.0 (UN) / Firmware: 1.1.4
  - Hardware: 1.0 (US) / Firmware: 1.1.13
- **LB110**
  - Hardware: 1.0 (US) / Firmware: 1.8.11

14.1.5 Light Strips

- **KL400L5**
  - Hardware: 1.0 (US) / Firmware: 1.0.5
  - Hardware: 1.0 (US) / Firmware: 1.0.8
- **KL420L5**
  - Hardware: 1.0 (US) / Firmware: 1.0.2
- **KL430**
  - Hardware: 2.0 (UN) / Firmware: 1.0.8
  - Hardware: 1.0 (US) / Firmware: 1.0.10
  - Hardware: 2.0 (US) / Firmware: 1.0.11
  - Hardware: 2.0 (US) / Firmware: 1.0.8
  - Hardware: 2.0 (US) / Firmware: 1.0.9

14.1.6 Hubs

- **KH100**
  - Hardware: 1.0 (UK) / Firmware: 1.5.6*

14.2 Tapo devices

All Tapo devices require authentication.

14.2.1 Plugs

- **P100**
  - Hardware: 1.0.0 / Firmware: 1.1.3
  - Hardware: 1.0.0 / Firmware: 1.3.7
  - Hardware: 1.0.0 / Firmware: 1.4.0
- **P110**
  - Hardware: 1.0 (EU) / Firmware: 1.0.7
  - Hardware: 1.0 (EU) / Firmware: 1.2.3
  - Hardware: 1.0 (UK) / Firmware: 1.3.0
- **P125M**
  - Hardware: 1.0 (US) / Firmware: 1.1.0
- **P135**
  - Hardware: 1.0 (US) / Firmware: 1.0.5
- **TP15**
  - Hardware: 1.0 (US) / Firmware: 1.0.3

14.2.2 Power Strips

- **P300**
  - Hardware: 1.0 (EU) / Firmware: 1.0.13
  - Hardware: 1.0 (EU) / Firmware: 1.0.7
- **TP25**
  - Hardware: 1.0 (US) / Firmware: 1.0.2

14.2.3 Wall Switches

- **S500D**
  - Hardware: 1.0 (US) / Firmware: 1.0.5
- **S505**
  - Hardware: 1.0 (US) / Firmware: 1.0.2

14.2.4 Bulbs

- **L510B**
  - Hardware: 3.0 (EU) / Firmware: 1.0.5
- **L510E**
  - Hardware: 3.0 (US) / Firmware: 1.0.5
  - Hardware: 3.0 (US) / Firmware: 1.1.2
- **L530E**
  - Hardware: 3.0 (EU) / Firmware: 1.0.6
  - Hardware: 3.0 (EU) / Firmware: 1.1.0
  - Hardware: 3.0 (EU) / Firmware: 1.1.6
  - Hardware: 2.0 (US) / Firmware: 1.1.0

14.2.5 Light Strips

- **L900-10**
  - Hardware: 1.0 (EU) / Firmware: 1.0.17
  - Hardware: 1.0 (US) / Firmware: 1.0.11
- **L900-5**
  - Hardware: 1.0 (EU) / Firmware: 1.0.17
  - Hardware: 1.0 (EU) / Firmware: 1.1.0
- **L920-5**
  - Hardware: 1.0 (US) / Firmware: 1.1.0
  - Hardware: 1.0 (US) / Firmware: 1.1.3
- **L930-5**
  - Hardware: 1.0 (US) / Firmware: 1.1.2

14.2.6 Hubs

- **H100**
  - Hardware: 1.0 (EU) / Firmware: 1.2.3
  - Hardware: 1.0 (EU) / Firmware: 1.5.5

PYTHON MODULE INDEX

k

INDEX

F

G

H

I

K

M

P

Q

R

S

T

U

V

X