HAP-python Documentation

Release 4...7...1

Ivan Kalchev

Jul 31, 2023

Contents

1 Getting Help **3**

2 First Steps **5**
2.1 HAP-python at a glance 5
2.2 Installation Guide 5
2.3 Tutorials 6
2.4 Examples 6

3 API Reference **7**
3.1 API Index 7

Python Module Index **19**

Index **21**

This documentation contains everything you need to know about HAP-python.

CHAPTER 1

Getting Help

Having Trouble? Post an issue on the GitHub repo, with as much information as possible about your issue.

CHAPTER 2

First Steps

2.1 HAP-python at a glance

HAP-python is an application framework to build out Accessories or Bridges for Apple's HomeKit protocol.

Need to fill in the rest.

2.2 Installation Guide

2.2.1 Before We Begin

HAP-python requires Python 3.4+. This guide will cover the current version of Raspbian and Ubuntu LTS. It is somewhat safe to assume the process for newer versions of Ubuntu will work.

2.2.2 Installing Pre-Requisites

Raspbian Stretch

As a prerequisite, you will need Avahi/Bonjour installed (due to `zeroconf` package):

```
sudo apt install libavahi-compat-libdnssd-dev
```

Ubuntu 16.04 LTS

Same with Raspbian, we will need to install Avahi/Bonjour, but a fresh 16.04 install will require the `python3-dev` package as well:

```
sudo apt install libavahi-compat-libdnssd-dev python3-dev
```

2.2.3 Installing HAP-python

Make a directory for your project, and `cd` into it:

```
~ $ mkdir hk_project
~ $ cd hk_project
~/hk_project $
```

It is best to use a virtualenv for most Python projects, we can use one here as well. Make sure that you have the `venv` module installed for Python 3:

```
sudo apt install python3-venv
```

To create a virtualenv and activate it, simply run these commands inside your project directory:

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

Because we used a Python 3 virtualenv and activated it, we can install `HAP-python` with `pip`:

```
pip install HAP-python
```

2.3 Tutorials

Need to fill in

2.4 Examples

Need to fill in

HAP-python at a glance Brief explanation of HAP-python, and the possible use cases.

Installation Guide How to install HAP-python on a Debian based system, such as a Raspberry Pi, or Ubuntu.

Tutorials Get started building your first HomeKit Accessory.

Examples A set of prebuilt accessories to either build your own class around, or to use as a starting point into your own custom Accessory class.

CHAPTER 3

API Reference

3.1 API Index

3.1.1 Accessory

Base class for HAP Accessories.

class `pyhap.accessory.`**`Accessory`**(*driver*, *display_name*, *aid=None*, *iid_manager=None*)

A representation of a HAP accessory.

Inherit from this class to build your own accessories.

`add_info_service`()

Helper method to add the required *AccessoryInformation* service.

Called in *__init__* to be sure that it is the first service added. May be overridden.

`add_preload_service`(*service*, *chars=None*, *unique_id=None*)

Create a service with the given name and add it to this acc.

`add_protocol_version_service`()

Helper method to add the required HAP Protocol Information service

`add_service`(**servs*)

Add the given services to this Accessory.

This also assigns unique IIDS to the services and their Characteristics.

Note: Do not add or remove characteristics from services that have been added to an Accessory, as this will lead to inconsistent IIDs.

Parameters **`servs`** – Variable number of services to add to this Accessory.

Type *Service*

available
Accessory is available.

If available is False, get_characteristics will return SERVICE_COMMUNICATION_FAILURE for the accessory which will show as unavailable.

Expected to be overridden.

get_characteristic(*aid*, *iid*)
Get the characteristic for the given IID.

The AID is used to verify if the search is in the correct accessory.

get_service(*name*)
Return a Service with the given name.

A single Service is returned even if more than one Service with the same name are present.

Parameters **name** (*str*) – The display_name of the Service to search for.

Returns A Service with the given name or None if no such service exists in this Accessory.

Return type *Service*

publish(*value*, *sender*, *sender_client_addr=None*, *immediate=False*)
Append AID and IID of the sender and forward it to the driver.

Characteristics call this method to send updates.

Parameters

- **data** (*dict*) – Data to publish, usually from a Characteristic.
- **sender** – The Service or Characteristic from which the call originated.

Type *Service* or *Characteristic*

run()
Called when the Accessory should start doing its thing.

Called when HAP server is running, advertising is set, etc. Can be overridden with a normal or async method.

static run_at_interval(*seconds*)
Decorator that runs decorated method every x seconds, until stopped.

Can be used with normal and async methods.

```
@Accessory.run_at_interval(3)
def run(self):
    print("Hello again world!")
```

Parameters **seconds** (*float*) – The amount of seconds to wait for the event to be set. Determines the interval on which the decorated method will be called.

set_info_service(*firmware_revision=None*, *manufacturer=None*, *model=None*, *serial_number=None*)
Quick assign basic accessory information.

set_primary_service(*primary_service*)
Set the primary service of the acc.

setup_message()
Print setup message to console.

For QRCode *base36*, *pyqrcode* are required. Installation through *pip install HAP-python[QRCode]*

stop()

Called when the Accessory should stop what is doing and clean up any resources.

Can be overridden with a normal or async method.

to_HAP()

A HAP representation of this Accessory.

Returns A HAP representation of this accessory. For example:

```
{ "aid": 1,
    "services": [{
        "iid" 2,
        "type": ...,
        ...
    }]
}
```

Return type dict

xhm_uri()

Generates the X-HM:// uri (Setup Code URI)

Return type str

3.1.2 AccessoryDriver

Accessory Driver class to host an Accessory.

class pyhap.accessory_driver.**AccessoryDriver**(**, address=None, port=51234, persist_file='accessory.state', pincode=None, encoder=None, loader=None, loop=None, mac=None, listen_address=None, advertised_address=None, interface_choice=None, async_zeroconf_instance=None, zeroconf_server=None*)

An AccessoryDriver mediates between incoming requests from the HAPServer and the Accessory.

The driver starts and stops the HAPServer, the mDNS advertisements and responds to events from the HAPServer.

accessories_hash

Hash the get_accessories response to track configuration changes.

add_accessory(*accessory*)

Add top level accessory to driver.

add_job(*target, *args*)

Add job to executor pool.

async_add_job(*target, *args*)

Add job from within the event loop.

async_persist()

Saves the state of the accessory.

Must be run in the event loop.

async_send_event (*topic*, *data*, *sender_client_addr*, *immediate*)
Send an event to a client.

Must be called in the event loop

async_start ()
Starts the accessory.

- Call the accessory's run method.
- Start handling accessory events.
- Start the HAP server.
- Publish a mDNS advertisement.
- Print the setup QR code if the accessory is not paired.

All of the above are started in separate threads. Accessory thread is set as daemon.

async_stop ()
Stops the AccessoryDriver and shutdown all remaining tasks.

async_subscribe_client_topic (*client*, *topic*, *subscribe=True*)
(Un)Subscribe the given client from the given topic.

This method must be run in the event loop.

Parameters

- **client** (`tuple <str, int>`) – A client (address, port) tuple that should be subscribed.
- **topic** (`str`) – The topic to which to subscribe.
- **subscribe** (`bool`) – Whether to subscribe or unsubscribe the client. Both subscribing an already subscribed client and unsubscribing a client that is not subscribed do nothing.

async_update_advertisement ()
Updates the mDNS service info for the accessory from the event loop.

config_changed ()
Notify the driver that the accessory's configuration has changed.

Persists the accessory, so that the new configuration is available on restart. Also, updates the mDNS advertisement, so that iOS clients know they need to fetch new data.

connection_lost (*client*)
Called when a connection is lost to a client.

This method must be run in the event loop.

Parameters client (`tuple <str, int>`) – A client (address, port) tuple that should be unsubscribed.

finish_pair ()
Finishing pairing or unpairing.

Updates the accessory and updates the mDNS service.

The mDNS announcement must not be updated until AFTER the final pairing response is sent or homekit will see that the accessory is already paired and assume it should stop pairing.

get_accessories ()
Returns the accessory in HAP format.

Returns An example HAP representation is:

```
{
   "accessories": [
      "aid": 1,
      "services": [
         "iid": 1,
         "type": ...,
         "characteristics": [{
            "iid": 2,
            "type": ...,
            "description": "CurrentTemperature",
            ...
         }]
      ]
   ]
}
```

Return type dict

get_characteristics(*char_ids*)
Returns values for the required characteristics.

Parameters **char_ids** (*list<str>*) – A list of characteristic "paths", e.g. "1.2" is aid 1, iid 2.

Returns Status success for each required characteristic. For example:

```
{
   "characteristics: [{
      "aid": 1,
      "iid": 2,
      "status" 0
   }]
}
```

Return type dict

load()
Load the persist file.

Must run in executor.

pair(*client_username_bytes: bytes*, *client_public: bytes*, *client_permissions: bytes*) → bool
Called when a client has paired with the accessory.

Persist the new accessory state.

Parameters

- **client_username_bytes** (*bytes*) – The client username bytes.
- **client_public** (*bytes*) – The client's public key.
- **client_permissions** (*bytes (int)*) – The client's permissions.

Returns Whether the pairing is successful.

Return type bool

persist()
Saves the state of the accessory.

Must run in executor.

prepare (*prepare_query*, *client_addr*)
Called from `HAPServerHandler` when iOS wants to prepare a write.

Parameters **prepare_query** – A prepare query. For example:

```
{
   "ttl": 10000, # in milliseconds
   "pid": 12345678,
}
```

publish (*data*, *sender_client_addr=None*, *immediate=False*)
Publishes an event to the client.

The publishing occurs only if the current client is subscribed to the topic for the aid and iid contained in the data.

Parameters **data** (`dict`) – The data to publish. It must at least contain the keys "aid" and "iid".

set_characteristics (*chars_query*, *client_addr*)
Called from `HAPServerHandler` when iOS configures the characteristics.

Parameters **chars_query** – A configuration query. For example:

```
{
   "characteristics": [{
      "aid": 1,
      "iid": 2,
      "value": False, # Value to set
      "ev": True # (Un)subscribe for events from this characteristics.
   }]
}
```

setup_srp_verifier ()
Create an SRP verifier for the accessory's info.

signal_handler (*_signal*, *_frame*)
Stops the AccessoryDriver for a given signal.

An AccessoryDriver can be registered as a signal handler with this method. For example, you can register it for a KeyboardInterrupt as follows: >>> import signal >>> signal.signal(signal.SIGINT, anAccDriver.signal_handler)

Now, when the user hits Ctrl+C, the driver will stop its accessory, the HAP server and everything else that needs stopping and will exit gracefully.

start ()
Start the event loop and call *start_service*.

Pyhap will be stopped gracefully on a KeyBoardInterrupt.

start_service ()
Start the service.

stop()
Method to stop pyhap.

unpair(*client_uuid*)
Removes the paired client from the accessory.

Persist the new accessory state.

Parameters **client_uuid** (`uuid.UUID`) – The client uuid.

update_advertisement()
Updates the mDNS service info for the accessory.

3.1.3 Bridge

Bridge Class to host multiple HAP Accessories.

class pyhap.accessory.**Bridge**(*driver*, *display_name*, *iid_manager=None*)
A representation of a HAP bridge.

A *Bridge* can have multiple *Accessories*.

add_accessory(*acc*)
Add the given `Accessory` to this `Bridge`.

Every `Accessory` in a `Bridge` must have an AID and this AID must be unique among all the `Accessories` in the same *Bridge*. If the given `Accessory`'s AID is None, a unique AID will be assigned to it. Otherwise, it will be verified that the AID is not the standalone aid (`STANDALONE_AID`) and that there is no other `Accessory` already in this `Bridge` with that AID.

Note: A `Bridge` cannot be added to another `Bridge`.

Parameters **acc** (`Accessory`) – The `Accessory` to be bridged.

Raises **ValueError** – When the given `Accessory` is of category `CATEGORY_BRIDGE` or if the AID of the `Accessory` clashes with another `Accessory` already in this `Bridge`.

get_characteristic(*aid*, *iid*)

See also:

Accessory.to_HAP

run()
Schedule tasks for each of the accessories' run method.

stop()
Calls stop() on all contained accessories.

to_HAP()
Returns a HAP representation of itself and all contained accessories.

See also:

Accessory.to_HAP

3.1.4 Characteristic

Characteristic Base class for a HAP Accessory `Service`.

See also:

pyhap.service.Service

class `pyhap.characteristic.`**`Characteristic`**(*display_name*, *type_id*, *properties*, *allow_invalid_client_values=False*, *unique_id=None*)

Represents a HAP characteristic, the smallest unit of the smart home.

A HAP characteristic is some measurement or state, like battery status or the current temperature. Characteristics are contained in services. Each characteristic has a unique type UUID and a set of properties, like format, min and max values, valid values and others.

`client_update_value`(*value*, *sender_client_addr=None*)

Called from broker for value change in Home app.

Change self.value to value and call callback.

classmethod `from_dict`(*name*, *json_dict*, *from_loader=False*)

Initialize a characteristic object from a dict.

Parameters **`json_dict`** (*`dict`*) – Dictionary containing at least the keys *Format*, *Permissions* and *UUID*

`get_value`()

This is to allow for calling *getter_callback*

Returns Current Characteristic Value

`notify`(*sender_client_addr=None*)

Notify clients about a value change. Sends the value.

See also:

accessory.publish

See also:

accessory_driver.publish

`override_properties`(*properties=None*, *valid_values=None*)

Override characteristic property values and valid values.

Parameters

- **`properties`** (*`dict`*) – Dictionary with values to override the existing properties. Only changed values are required.
- **`valid_values`** (*`dict`*) – Dictionary with values to override the existing valid_values. Valid values will be set to new dictionary.

`set_value`(*value*, *should_notify=True*)

Set the given raw value. It is checked if it is a valid value.

If not set_value will be aborted and an error message will be displayed.

Characteristic.setter_callback You may also define a *setter_callback* on the *Characteristic*. This will be called with the value being set as the arg.

See also:

Characteristic.value

Parameters

- **`value`** (*`Depends on properties["Format"]`*) – The value to assign as this Characteristic's value.
- **`should_notify`** (*`bool`*) – Whether a the change should be sent to subscribed clients. Notify will be performed if the broker is set.

`to_HAP``()`
Create a HAP representation of this Characteristic.

Used for json serialization.

Returns A HAP representation.

Return type dict

`to_valid_value``(`*value*`)`
Perform validation and conversion to valid value.

`valid_value_or_raise``(`*value*`)`
Raise ValueError if PROP_VALID_VALUES is set and the value is not present.

3.1.5 Loader

Useful for creating a `Service` or `Characteristic`.

`class` `pyhap.loader.`**`Loader`**`(`*path_char='/home/docs/checkouts/readthedocs.org/user_builds/hap-python/checkouts/stable/pyhap/resources/characteristics.json'*, *path_service='/home/docs/checkouts/readthedocs.org/user_builds/hap-python/checkouts/stable/pyhap/resources/services.json'*`)`
Looks up type descriptions based on a name.

See also:

pyhap/resources/services.json

See also:

pyhap/resources/characteristics.json

`classmethod from_dict``(`*char_dict=None*, *serv_dict=None*`)`
Create a new instance directly from json dicts.

`get_char``(`*name*`)`
Return new Characteristic object.

`get_service``(`*name*`)`
Return new service object.

3.1.6 Service

Service Base class for a HAP `Accessory`.

`class` `pyhap.service.`**`Service`**`(`*type_id*, *display_name=None*, *unique_id=None*`)`
A representation of a HAP service.

A Service contains multiple characteristics. For example, a TemperatureSensor service has the characteristic CurrentTemperature.

add_characteristic(**chars*)
Add the given characteristics as "mandatory" for this Service.

add_linked_service(*service*)
Add the given service as "linked" to this Service.

configure_char(*char_name, properties=None, valid_values=None, value=None, setter_callback=None, getter_callback=None*)
Helper method to return fully configured characteristic.

classmethod from_dict(*name, json_dict, loader*)
Initialize a service object from a dict.

Parameters **json_dict** (`dict`) – Dictionary containing at least the keys *UUID* and *RequiredCharacteristics*

get_characteristic(*name*)
Return a Characteristic object by the given name from this Service.

Parameters **name** (`str`) – The name of the characteristic to search for.

:raise ValueError if characteristic is not found.

Returns A characteristic with the given name.

Return type *Characteristic*

to_HAP()
Create a HAP representation of this Service.

Returns A HAP representation.

Return type dict.

3.1.7 State

class pyhap.state.**State**(**, address: Union[str, List[str], None] = None, mac=None, pincode=None, port=None*)
Class to store all (semi-)static information.

That includes all needed for setup of driver and pairing.

add_paired_client(*client_username_bytes: bytes, client_public: bytes, perms: bytes*) → None
Add a given client to dictionary of paired clients.

Parameters

- **client_username_bytes** (`bytes`) – The client's user id bytes.
- **client_public** (`bytes`) – The client's public key (not the session public key).

address
Return the first address for backwards compat.

increment_config_version()
Increment the config version.

is_admin(*client_uuid: uuid.UUID*) → bool
Check if a paired client is an admin.

paired
Return if main accessory is currently paired.

remove_paired_client (*client_uuid: uuid.UUID*) → None
Remove a given client from dictionary of paired clients.

Parameters **client_uuid** (`uuid.UUID`) – The client's UUID.

set_accessories_hash (*accessories_hash*)
Set the accessories hash and increment the config version if needed.

3.1.8 Util

Utilities Module

pyhap.util.**base64_to_bytes** (*str_input*) → bytes

pyhap.util.**byte_bool** (*boolv*)

pyhap.util.**callback** (*func*)
Decorator for non blocking functions.

pyhap.util.**event_wait** (*event*, *timeout*)
Wait for the given event to be set or for the timeout to expire.

Parameters

- **event** (`asyncio.Event`) – The event to wait for.
- **timeout** (`float`) – The timeout for which to wait, in seconds.

Returns `event.is_set()`

Return type bool

pyhap.util.**from_hap_json** (*json_str*)
Convert json to an object.

pyhap.util.**generate_mac** ()
Generates a fake mac address used in broadcast.

Returns MAC address in format XX:XX:XX:XX:XX:XX

Return type str

pyhap.util.**generate_pincode** ()
Generates a random pincode.

Returns pincode in format `xxx-xx-xxx`

Return type bytearray

pyhap.util.**generate_setup_id** ()
Generates a random Setup ID for an `Accessory` or `Bridge`.

Used in QR codes and the setup hash.

Returns 4 digit alphanumeric code.

Return type str

pyhap.util.**get_local_address** () → str
Grabs the local IP address using a socket.

Returns Local IP Address in IPv4 format.

Return type str

`pyhap.util.`**`hap_type_to_uuid`**
: Convert a HAP type to a UUID.

`pyhap.util.`**`is_callback`**(*func*)
: Check if function is callback.

`pyhap.util.`**`iscoro`**(*func*)
: Check if the function is a coroutine or if the function is a `functools.partial`, check the wrapped function for the same.

`pyhap.util.`**`long_to_bytes`**(*n*)
: Convert a `long int` to `bytes`

    Parameters **n** (*int*) – Long Integer

    Returns `long int` in `bytes` format.

    Return type bytes

`pyhap.util.`**`to_base64_str`**(*bytes_input*) → str

`pyhap.util.`**`to_hap_json`**(*dump_obj*)
: Convert an object to HAP json.

`pyhap.util.`**`to_sorted_hap_json`**(*dump_obj*)
: Convert an object to sorted HAP json.

`pyhap.util.`**`uuid_to_hap_type`**
: Convert a UUID to a HAP type.

API Index API documentation for HAP-python.

Python Module Index

p

pyhap.util, 17

Index

A

accessories_hash (*pyhap.accessory_driver.AccessoryDriver attribute*), 9
Accessory (*class in pyhap.accessory*), 7
AccessoryDriver (*class in pyhap.accessory_driver*), 9
add_accessory() (*pyhap.accessory.Bridge method*), 13
add_accessory() (*pyhap.accessory_driver.AccessoryDriver method*), 9
add_characteristic() (*pyhap.service.Service method*), 15
add_info_service() (*pyhap.accessory.Accessory method*), 7
add_job() (*pyhap.accessory_driver.AccessoryDriver method*), 9
add_linked_service() (*pyhap.service.Service method*), 16
add_paired_client() (*pyhap.state.State method*), 16
add_preload_service() (*pyhap.accessory.Accessory method*), 7
add_protocol_version_service() (*pyhap.accessory.Accessory method*), 7
add_service() (*pyhap.accessory.Accessory method*), 7
address (*pyhap.state.State attribute*), 16
async_add_job() (*pyhap.accessory_driver.AccessoryDriver method*), 9
async_persist() (*pyhap.accessory_driver.AccessoryDriver method*), 9
async_send_event() (*pyhap.accessory_driver.AccessoryDriver method*), 9
async_start() (*pyhap.accessory_driver.AccessoryDriver method*), 10
async_stop() (*pyhap.accessory_driver.AccessoryDriver method*), 10
async_subscribe_client_topic() (*pyhap.accessory_driver.AccessoryDriver method*), 10
async_update_advertisement() (*pyhap.accessory_driver.AccessoryDriver method*), 10
available (*pyhap.accessory.Accessory attribute*), 7

B

base64_to_bytes() (*in module pyhap.util*), 17
Bridge (*class in pyhap.accessory*), 13
byte_bool() (*in module pyhap.util*), 17

C

callback() (*in module pyhap.util*), 17
Characteristic (*class in pyhap.characteristic*), 14
client_update_value() (*pyhap.characteristic.Characteristic method*), 14
config_changed() (*pyhap.accessory_driver.AccessoryDriver method*), 10
configure_char() (*pyhap.service.Service method*), 16
connection_lost() (*pyhap.accessory_driver.AccessoryDriver method*), 10

E

event_wait() (*in module pyhap.util*), 17

F

finish_pair() (*pyhap.accessory_driver.AccessoryDriver method*), 10

from_dict() (*pyhap.characteristic.Characteristic class method*), 14
from_dict() (*pyhap.loader.Loader class method*), 15
from_dict() (*pyhap.service.Service class method*), 16
from_hap_json() (*in module pyhap.util*), 17

G

generate_mac() (*in module pyhap.util*), 17
generate_pincode() (*in module pyhap.util*), 17
generate_setup_id() (*in module pyhap.util*), 17
get_accessories() (*pyhap.accessory_driver.AccessoryDriver method*), 10
get_char() (*pyhap.loader.Loader method*), 15
get_characteristic() (*pyhap.accessory.Accessory method*), 8
get_characteristic() (*pyhap.accessory.Bridge method*), 13
get_characteristic() (*pyhap.service.Service method*), 16
get_characteristics() (*pyhap.accessory_driver.AccessoryDriver method*), 11
get_local_address() (*in module pyhap.util*), 17
get_service() (*pyhap.accessory.Accessory method*), 8
get_service() (*pyhap.loader.Loader method*), 15
get_value() (*pyhap.characteristic.Characteristic method*), 14

H

hap_type_to_uuid (*in module pyhap.util*), 17

I

increment_config_version() (*pyhap.state.State method*), 16
is_admin() (*pyhap.state.State method*), 16
is_callback() (*in module pyhap.util*), 18
iscoro() (*in module pyhap.util*), 18

L

load() (*pyhap.accessory_driver.AccessoryDriver method*), 11
Loader (*class in pyhap.loader*), 15
long_to_bytes() (*in module pyhap.util*), 18

N

notify() (*pyhap.characteristic.Characteristic method*), 14

O

override_properties() (*pyhap.characteristic.Characteristic method*), 14

P

pair() (*pyhap.accessory_driver.AccessoryDriver method*), 11
paired (*pyhap.state.State attribute*), 16
persist() (*pyhap.accessory_driver.AccessoryDriver method*), 11
prepare() (*pyhap.accessory_driver.AccessoryDriver method*), 12
publish() (*pyhap.accessory.Accessory method*), 8
publish() (*pyhap.accessory_driver.AccessoryDriver method*), 12
pyhap.util (*module*), 17

R

remove_paired_client() (*pyhap.state.State method*), 16
run() (*pyhap.accessory.Accessory method*), 8
run() (*pyhap.accessory.Bridge method*), 13
run_at_interval() (*pyhap.accessory.Accessory static method*), 8

S

Service (*class in pyhap.service*), 15
set_accessories_hash() (*pyhap.state.State method*), 17
set_characteristics() (*pyhap.accessory_driver.AccessoryDriver method*), 12
set_info_service() (*pyhap.accessory.Accessory method*), 8
set_primary_service() (*pyhap.accessory.Accessory method*), 8
set_value() (*pyhap.characteristic.Characteristic method*), 14
setup_message() (*pyhap.accessory.Accessory method*), 8
setup_srp_verifier() (*pyhap.accessory_driver.AccessoryDriver method*), 12
signal_handler() (*pyhap.accessory_driver.AccessoryDriver method*), 12
start() (*pyhap.accessory_driver.AccessoryDriver method*), 12
start_service() (*pyhap.accessory_driver.AccessoryDriver method*), 12
State (*class in pyhap.state*), 16
stop() (*pyhap.accessory.Accessory method*), 9
stop() (*pyhap.accessory.Bridge method*), 13
stop() (*pyhap.accessory_driver.AccessoryDriver method*), 12

T

to_base64_str() (*in module pyhap.util*), 18
to_HAP() (*pyhap.accessory.Accessory method*), 9
to_HAP() (*pyhap.accessory.Bridge method*), 13
to_HAP() (*pyhap.characteristic.Characteristic method*), 15
to_HAP() (*pyhap.service.Service method*), 16
to_hap_json() (*in module pyhap.util*), 18
to_sorted_hap_json() (*in module pyhap.util*), 18
to_valid_value() (*pyhap.characteristic.Characteristic method*), 15

U

unpair() (*pyhap.accessory_driver.AccessoryDriver method*), 13
update_advertisement() (*pyhap.accessory_driver.AccessoryDriver method*), 13
uuid_to_hap_type (*in module pyhap.util*), 18

V

valid_value_or_raise() (*pyhap.characteristic.Characteristic method*), 15

X

xhm_uri() (*pyhap.accessory.Accessory method*), 9