Libpurecoollink Documentation

Release 0.1.0

Charles Blonde

Jul 16, 2017

Contents

This Python 3.4+ library allow you to control Dyson fan/purifier devices and Dyson 360 Eye robot vacuum device.

CHAPTER 1

Status

Backward compatibility is a goal but breaking changes can still happen.

Discovery is not fully reliable yet. It's working most of the time but sometimes discovery will not work. Manual configuration is available to bypass this limitation.

Supported devices

- Dyson pure cool link devices (Tower and Desk)
- Dyson Cool+Hot devices
- Dyson 360 Eye robot vacuum

CHAPTER 2

Features

Commands

The following commands are supported:

- Purifier/fan devices
  - Connect to the device using discovery or manually with IP Address
  - Turn on/off
  - Set speed
  - Turn on/off oscillation
  - Set Auto mode
  - Set night mode
  - Set sleep timer
  - Set Air Quality target (Normal, High, Better)
  - Enable/disable standby monitoring (the device continue to update sensors when in standby)
  - Reset filter life
- Cool+Hot purifier/fan devices
  - Set heat mode
  - Set heat target
  - Set fan focus mode
- 360 Eye device
  - Set power mode (Quiet/Max)
  - Start cleaning
  - Pause cleaning

  - Resume cleaning
  - Abort cleaning

Sensors

The following sensors are available for fan/purifier devices:

- Humidity
- Temperature in Kelvin
- Dust (unknown metric)
- Air Quality (unknown metric)

CHAPTER 3

Usage

Installation

```
pip install libpurecoollink
```

Dyson account

In order to access the devices, you need to have access to a valid Dyson account.

```
from libpurecoollink.dyson import DysonAccount

# Log to Dyson account
# Language is a two characters code (eg: FR)
dyson_account = DysonAccount("<dyson_account_email>","<dyson_account_password>","
↪<language>")
logged = dyson_account.login()
```

Fan/Purifier devices

Connect to devices

After login to the Dyson account, known devices are available.

Connections to the devices can been done automatically using mDNS or manually with specifying IP address

Automatic connection (mDNS)

```
from libpurecoollink.dyson import DysonAccount

# Log to Dyson account
# Language is a two characters code (eg: FR)
dyson_account = DysonAccount("<dyson_account_email>","<dyson_account_password>","
↪<language>")
logged = dyson_account.login()

if not logged:
    print('Unable to login to Dyson account')
    exit(1)

# List devices available on the Dyson account
devices = dyson_account.devices()

# Connect using discovery to the first device
connected = devices[0].connect()

# connected == device available, state values are available, sensor values are␣
↪available
```

Manual connection

```
from libpurecoollink.dyson import DysonAccount

# Log to Dyson account
# Language is a two characters code (eg: FR)
dyson_account = DysonAccount("<dyson_account_email>","<dyson_account_password>","
↪<language>")
logged = dyson_account.login()

if not logged:
    print('Unable to login to Dyson account')
    exit(1)

# List devices available on the Dyson account
devices = dyson_account.devices()

# Connect using discovery to the first device
connected = devices[0].connect(device_ip="192.168.1.2")

# connected == device available, state values are available, sensor values are␣
↪available
```

Disconnect from the device

Disconnection is required for fan/purifier devices in order to release resources (an internal thread is started to request update notifications)

```
from libpurecoollink.dyson import DysonAccount

# ... connection do dyson account and to device ... #
```

```
# Disconnect
devices[0].disconnect()
```

Send commands

After connected to the device, commands cand be send in order to update the device configuration

```
from libpurecoollink.dyson import DysonAccount
from libpurecoollink.const import FanSpeed, FanMode, NightMode, Oscillation, \
    FanState, StandbyMonitoring, QualityTarget, ResetFilter, HeatMode, \
    FocusMode, HeatTarget

# ... connection do dyson account and to device ... #

# Turn on the fan to speed 2
devices[0].set_configuration(fan_mode=FanMode.FAN, fan_speed=FanSpeed.FAN_SPEED_2)

# Turn on oscillation
devices[0].set_configuration(oscillation=Oscillation.OSCILLATION_ON)

# Turn on night mode
devices[0].set_configuration(night_mode=NightMode.NIGHT_MODE_ON)

# Set 10 minutes sleep timer
devices[0].set_configuration(sleep_timer=10)

# Disable sleep timer
devices[0].set_configuration(sleep_timer=0)

# Set quality target (for auto mode)
devices[0].set_configuration(quality_target=QualityTarget.QUALITY_NORMAL)

# Disable standby monitoring
devices[0].set_configuration(standby_monitoring=StandbyMonitoring.STANDBY_MONITORING_
↪OFF)

# Reset filter life
devices[0].set_configuration(reset_filter=ResetFilter.RESET_FILTER)

## Cool+Hot devices only
# Set Heat mode
devices[0].set_configuration(heat_mode=HeatMode.HEAT_ON)
# Set heat target
devices[0].set_configuration(heat_target=HeatTarget.celsius(25))
devices[0].set_configuration(heat_target=HeatTarget.fahrenheit(70))
# Set fan focus mode
devices[0].set_configuration(focus_mode=FocusMode.FOCUS_ON)

# Everything can be mixed in one call
devices[0].set_configuration(sleep_timer=10,
    fan_mode=FanMode.FAN,
    fan_speed=FanSpeed.FAN_SPEED_5,
    night_mode=NightMode.NIGHT_MODE_OFF,
    standby_monitoring=StandbyMonitoring.STANDBY_MONITORING_ON,
    quality_target=QualityTarget.QUALITY_HIGH)
```

States and sensors

States and sensors values are available using *state* and *environment_state* properties

States values

```
# ... imports ... #

# ... connection do dyson account and to device ... #

print(devices[0].state.speed)
print(devices[0].state.oscillation)
# ... #
```

Environmental values

```
# ... imports ... #

# ... connection do dyson account and to device ... #

print(devices[0].environment_state.humidity)
print(devices[0].environment_state.sleep_timer)
# ... #
```

All properties are available in the sources.

Notifications

You can register to any values changed by using a callback function

```
# ... imports ... #
from libpurecoollink.dyson_pure_state import DysonPureHotCoolState, \
  DysonPureCoolState, DysonEnvironmentalSensorState

# ... connection do dyson account and to device ... #
def on_message(msg):
    # Message received
    if isinstance(msg, DysonPureCoolState):
        # Will be true for DysonPureHotCoolState too.
        print("DysonPureCoolState message received")
    if isinstance(msg, DysonPureHotCoolState):
        print("DysonPureHotCoolState message received")
    if isinstance(msg, DysonEnvironmentalSensorState)
        print("DysonEnvironmentalSensorState received")
    print(msg)

devices[0].connect()
devices[0].add_message_listener(on_message)
```

360 Eye robot vacuum

Connect to devices

After login to the Dyson account, known devices are available.

Auto discovery is not yet supported.

Manual connection

```
from libpurecoollink.dyson import DysonAccount

# Log to Dyson account
# Language is a two characters code (eg: FR)
dyson_account = DysonAccount("<dyson_account_email>","<dyson_account_password>","
↪<language>")
logged = dyson_account.login()

if not logged:
    print('Unable to login to Dyson account')
    exit(1)

# List devices available on the Dyson account
devices = dyson_account.devices()

# Connect using discovery to the first device
connected = devices[0].connect("192.168.1.2")

# connected == device available, state values are available, sensor values are␣
↪available
```

Send commands

After connected to the device, commands cand be send in order to update the device configuration.

```
import time
from libpurecoollink.dyson import DysonAccount
from libpurecoollink.const import PowerMode

# ... connection do dyson account and to device ... #

# Set power mode
devices[0].set_power_mode(PowerMode.QUIET)
devices[0].set_power_mode(PowerMode.MAX)

# Start cleaning
devices[0].start

time.sleep(30)

# Pause cleaning
devices[0].pause()

time.sleep(30)

# Resume cleaning
devices[0].resume()

time.sleep(30)

# Abort cleaning (device return to the base)
devices[0].abort()
```

States

State values are available using *state* property.

```
# ... imports ... #

# ... connection do dyson account and to device ... #

print(devices[0].state.state)
print(devices[0].state.full_clean_type)
print(devices[0].state.position)
print(devices[0].state.power_mode)
print(devices[0].state.battery_level)
print(devices[0].state.clean_id)
# ... #
```

All properties are available in the sources.

Notifications

You can register to any values changed by using a callback function

```
# ... imports ... #
from libpurecoollink.dyson_360_eye import Dyson360EyeState, \
  Dyson360EyeTelemetryData, Dyson360EyeMapData, Dyson360EyeMapGrid, \
  Dyson360EyeMapGlobal

# ... connection do dyson account and to device ... #
def on_message(msg):
    # Message received
    if isinstance(msg, Dyson360EyeState):
        print("Dyson360EyeState message received")
    if isinstance(msg, Dyson360EyeTelemetryData)
        print("Dyson360EyeTelemetryData received")
    if isinstance(msg, Dyson360EyeMapData)
        print("Dyson360EyeMapData received")
    if isinstance(msg, Dyson360EyeMapGrid)
        print("Dyson360EyeMapGrid received")
    if isinstance(msg, Dyson360EyeMapGlobal)
        print("Dyson360EyeMapGlobal received")
    print(msg)

devices[0].connect()
devices[0].add_message_listener(on_message)
```

CHAPTER 4

API Documentation

If you are looking for information on a specific function, class, or method, this part of the documentation is for you.

Developer Interface

This part of the documentation covers all the interfaces of Libpurecoollink.

Classes

Common

DysonAccount

class libpurecoollink.dyson.**DysonAccount**(*email*, *password*, *country*)
Dyson account.

devices()
Return all devices linked to the account.

logged
Return True if user is logged, else False.

login()
Login to dyson web services.

NetworkDevice

class libpurecoollink.dyson_device.**NetworkDevice**(*name*, *address*, *port*)
Network device.

address
Device address.

name
Device name.

port
Device port.

Fan/Purifier devices

DysonPureCoolLink

class libpurecoollink.dyson_pure_cool_link.**DysonPureCoolLink**(*json_body*)
Dyson device (fan).

class **DysonDeviceListener**(*serial*, *add_device_function*)
Message listener.

add_service(*zeroconf*, *device_type*, *name*)
Add device.

Parameters
- **zeroconf** – MSDNS object
- **device_type** – Service type
- **name** – Device name

remove_service(*zeroconf*, *device_type*, *name*)
Remove listener.

DysonPureCoolLink.**active**
Active status.

DysonPureCoolLink.**add_message_listener**(*callback_message*)
Add message listener.

DysonPureCoolLink.**auto_update**
Auto update configuration.

DysonPureCoolLink.**callback_message**
Return callback functions when message are received.

DysonPureCoolLink.**clear_message_listener**()
Clear all message listener.

DysonPureCoolLink.**command_topic**
MQTT command topic.

DysonPureCoolLink.**connect**(*on_message=None*, *device_ip=None*, *timeout=5*, *retry=15*)
Try to connect to device.

If device_ip is provided, mDNS discovery step will be skipped.

Parameters
- **on_message** – On Message callback function
- **device_ip** – Device IP address
- **timeout** – Timeout
- **retry** – Max retry

DysonPureCoolLink.**connected**
Device connected.

DysonPureCoolLink.**connection_callback**(*connected*)
Set function called when device is connected.

DysonPureCoolLink.**credentials**
Device encrypted credentials.

DysonPureCoolLink.**device_available**
Return True if device is fully available, else false.

DysonPureCoolLink.**disconnect**()
Disconnect from the device.

DysonPureCoolLink.**environmental_state**
Environmental Device state.

DysonPureCoolLink.**name**
Device name.

DysonPureCoolLink.**network_device**
Network device.

DysonPureCoolLink.**new_version_available**
Return if new version available.

DysonPureCoolLink.**on_connect**(*client*, *userdata*, *flags*, *return_code*)
Set function callback when connected.

static DysonPureCoolLink.**on_message**(*client*, *userdata*, *msg*)
Set function Callback when message received.

DysonPureCoolLink.**product_type**
Product type.

DysonPureCoolLink.**remove_message_listener**(*callback_message*)
Remove a message listener.

DysonPureCoolLink.**request_current_state**()
Request new state message.

DysonPureCoolLink.**request_environmental_state**()
Request new state message.

DysonPureCoolLink.**sensor_data_available**()
Call when first sensor data are available. Internal method.

DysonPureCoolLink.**serial**
Device serial.

DysonPureCoolLink.**set_configuration**(***kwargs*)
Configure fan.

Parameters **kwargs** – Parameters

DysonPureCoolLink.**set_fan_configuration**(*data*)
Configure Fan.

Parameters **data** – Data to send

DysonPureCoolLink.**state**
Device state.

DysonPureCoolLink.**state_data_available**()
Call when first state data are available. Internal method.

DysonPureCoolLink.**status_topic**
MQTT status topic.

DysonPureCoolLink.**version**
Device version.

DysonPureHotCoolLink

class libpurecoollink.dyson_pure_hotcool_link.**DysonPureHotCoolLink**(*json_body*)
Dyson Pure Hot+Cool device.

class **DysonDeviceListener**(*serial*, *add_device_function*)
Message listener.

add_service(*zeroconf*, *device_type*, *name*)
Add device.

Parameters

- **zeroconf** – MSDNS object
- **device_type** – Service type
- **name** – Device name

remove_service(*zeroconf*, *device_type*, *name*)
Remove listener.

DysonPureHotCoolLink.**active**
Active status.

DysonPureHotCoolLink.**add_message_listener**(*callback_message*)
Add message listener.

DysonPureHotCoolLink.**auto_update**
Auto update configuration.

DysonPureHotCoolLink.**callback_message**
Return callback functions when message are received.

DysonPureHotCoolLink.**clear_message_listener**()
Clear all message listener.

DysonPureHotCoolLink.**command_topic**
MQTT command topic.

DysonPureHotCoolLink.**connect**(*on_message=None*, *device_ip=None*, *timeout=5*, *retry=15*)
Try to connect to device.

If device_ip is provided, mDNS discovery step will be skipped.

Parameters

- **on_message** – On Message callback function
- **device_ip** – Device IP address
- **timeout** – Timeout
- **retry** – Max retry

DysonPureHotCoolLink.**connected**
Device connected.

DysonPureHotCoolLink.**connection_callback**(*connected*)
Set function called when device is connected.

DysonPureHotCoolLink.**credentials**
Device encrypted credentials.

DysonPureHotCoolLink.**device_available**
Return True if device is fully available, else false.

DysonPureHotCoolLink.**disconnect**()
Disconnect from the device.

DysonPureHotCoolLink.**environmental_state**
Environmental Device state.

DysonPureHotCoolLink.**name**
Device name.

DysonPureHotCoolLink.**network_device**
Network device.

DysonPureHotCoolLink.**new_version_available**
Return if new version available.

DysonPureHotCoolLink.**on_connect**(*client*, *userdata*, *flags*, *return_code*)
Set function callback when connected.

DysonPureHotCoolLink.**on_message**(*client*, *userdata*, *msg*)
Set function Callback when message received.

DysonPureHotCoolLink.**product_type**
Product type.

DysonPureHotCoolLink.**remove_message_listener**(*callback_message*)
Remove a message listener.

DysonPureHotCoolLink.**request_current_state**()
Request new state message.

DysonPureHotCoolLink.**request_environmental_state**()
Request new state message.

DysonPureHotCoolLink.**sensor_data_available**()
Call when first sensor data are available. Internal method.

DysonPureHotCoolLink.**serial**
Device serial.

DysonPureHotCoolLink.**set_configuration**(***kwargs*)
Configure fan.

Parameters **kwargs** – Parameters

DysonPureHotCoolLink.**set_fan_configuration**(*data*)
Configure Fan.

Parameters **data** – Data to send

DysonPureHotCoolLink.**state**
Device state.

DysonPureHotCoolLink.**state_data_available**()
Call when first state data are available. Internal method.

DysonPureHotCoolLink.**status_topic**
MQTT status topic.

DysonPureHotCoolLink.**version**
Device version.

DysonPureCoolState

class libpurecoollink.dyson_pure_state.**DysonPureCoolState**(*payload*)
Dyson device state.

fan_mode
Fan mode.

fan_state
Fan state.

filter_life
Filter life.

static is_state_message(*payload*)
Return true if this message is a Dyson Pure state message.

night_mode
Night mode.

oscillation
Oscillation mode.

quality_target
Air quality target.

speed
Fan speed.

standby_monitoring
Monitor when inactive (standby).

DysonEnvironmentalSensorState

class libpurecoollink.dyson_pure_state.**DysonEnvironmentalSensorState**(*payload*)
Environmental sensor state.

dust
Dust level.

humidity
Humidity in percent.

static is_environmental_state_message(*payload*)
Return true if this message is a state message.

sleep_timer
Sleep timer.

temperature
Temperature in Kelvin.

volatil_organic_compounds
Volatil organic compounds level.

DysonPureHotCoolState

class libpurecoollink.dyson_pure_state.**DysonPureHotCoolState**(*payload*)
Dyson device state.

fan_mode
Fan mode.

fan_state
Fan state.

filter_life
Filter life.

focus_mode
Focus the fan on one stream or spread.

heat_mode
Heat mode on or off.

heat_state
Return heat state.

heat_target
Heat target of the temperature.

is_state_message(*payload*)
Return true if this message is a Dyson Pure state message.

night_mode
Night mode.

oscillation
Oscillation mode.

quality_target
Air quality target.

speed
Fan speed.

standby_monitoring
Monitor when inactive (standby).

tilt
Return tilt status.

Eye 360 robot vacuum device

Dyson360Eye

class libpurecoollink.dyson_360_eye.**Dyson360Eye**(*json_body*)
Dyson 360 Eye device.

abort()
Abort cleaning.

active
Active status.

add_message_listener(*callback_message*)
Add message listener.

auto_update
Auto update configuration.

static call_callback_functions(*functions*, *message*)
Call callback functions.

callback_message
Return callback functions when message are received.

clear_message_listener()
Clear all message listener.

command_topic
MQTT command topic.

connect(*device_ip*)
Try to connect to device.

Parameters **device_ip** – Device IP address

connection_callback(*connected*)
Set function called when device is connected.

credentials
Device encrypted credentials.

device_available
Return True if device is fully available, else false.

name
Device name.

network_device
Network device.

new_version_available
Return if new version available.

on_connect(*client*, *userdata*, *flags*, *return_code*)
Set function callback when connected.

static on_message(*client*, *userdata*, *msg*)
Set function Callback when message received.

pause()
Pause cleaning.

product_type
Product type.

remove_message_listener(*callback_message*)
Remove a message listener.

request_current_state()
Request new state message.

resume()
Resume cleaning.

serial
Device serial.

set_power_mode(*power_mode*)
Set power mode.

:param power_mode Power mode (const.PowerMode)

start()
Start cleaning.

state
Device state.

state_data_available()
Call when first state data are available. Internal method.

status_topic
MQTT status topic.

version
Device version.

Dyson360EyeState

class libpurecoollink.dyson_360_eye.**Dyson360EyeState**(*json_body*)
Dyson 360 Eye state.

battery_level
Return battery level.

clean_id
Return clean id.

full_clean_type
Return full clean type.

static is_state_message(*payload*)
Return true if this message is a Dyson 360 Eye state message.

position
Return position.

power_mode
Return power mode.

state
Return state status.

Dyson360EyeTelemetryData

class libpurecoollink.dyson_360_eye.**Dyson360EyeTelemetryData**(*json_body*)
Dyson 360 Eye Telemetry Data.

field1
Return field 1.

field2
Return field 2.

field3
Return field 3.

field4
Return field 4.

static is_telemetry_data(*payload*)
Return true if this message is a telemetry data message.

telemetry_data_id
Return Telemetry data id.

time
Return time.

Dyson360EyeMapData

class libpurecoollink.dyson_360_eye.**Dyson360EyeMapData**(*json_body*)
Dyson 360 Eye map data.

clean_id
Return Clean Id.

content
Return content.

content_encoding
Return content encoding.

content_type
Return content type.

grid_id
Return Grid id.

static is_map_data(*payload*)
Return true if this message is a map data message.

time
Return time.

Dyson360EyeMapGrid

class libpurecoollink.dyson_360_eye.**Dyson360EyeMapGrid**(*json_body*)
Dyson 360 Eye map grid.

anchor
Return Anchor.

clean_id
Return clean id.

grid_id
Return grid id.

height
Return height.

static is_map_grid(*payload*)
Return true if this message is a map grid message.

resolution
Return resolution.

time
Return time.

width
Return width.

Dyson360EyeMapGlobal

class libpurecoollink.dyson_360_eye.**Dyson360EyeMapGlobal**(*json_body*)
Dyson 360Eye map global.

angle
Return angle.

clean_id
Return clean id.

grid_id
Return grid id.

static is_map_global(*payload*)
Return true if this message is a map global message.

position_x
Return x.

position_y
Return y.

time
Return time.

Exceptions

DysonNotLoggedException

exception libpurecoollink.exceptions.**DysonNotLoggedException**
Not logged to Dyson Web Services Exception.

DysonInvalidTargetTemperatureException

exception libpurecoollink.exceptions.**DysonInvalidTargetTemperatureException**(*temperature_unit*, *current_value*)
Invalid target temperature Exception.

CHAPTER 5

How it's working

Dyson devices use many different protocols in order to work:

- HTTPS to Dyson API in order to get devices informations (credentials, historical data, etc ...)
- MDNS to discover devices on the local network
- MQTT (with auth) to get device status and send commands

To my knowledge, no public technical information about API/MQTT are available so all the work is done by testing and a lot of properties are unknown to me at this time.

This library come with a modified version of Zeroconf because Dyson MDNS implementation is not valid.

This documentation help me to understand some of return values.

CHAPTER 6

Work to do

- Better protocol understanding
- Better technical documentation on how it is working
- Get historical data from the API (air quality, etc ...)

CHAPTER 7

Releases

Versions

Version 0.3.0

Date 2017/07/08

- Add support for heating devices
- Add reset filter life feature

Version 0.2.0

Date 2017/06/18

- First official Pypi release

CHAPTER 8

Contributors

Thanks to all the wonderful folks who have contributed to Libpurecoollink:

- ThomasHoussin <https://github.com/ThomasHoussin> (add parameters)
- Soraxas <https://github.com/soraxas> Add Cool+Hot devices support

Python Module Index

l

Index

D

E

F

P

Q

R

S

T

V

W