pyremoteplay Documentation

Release 0.7.6

ktnrg45

Aug 23, 2022

TABLE OF CONTENTS:

CHAPTER

ONE

OVERVIEW

1.1 pyremoteplay

Python PlayStation Remote Play API

Documentation

1.1.1 About

This project provides an API to programmatically connect to and control Remote Play hosts (PS4 and PS5). The low-level networking internals is written using the Asyncio framework. In addition it includes an optional GUI, allowing to view the live stream and control the host through keyboard/mouse input. This library is based on the C/C++ project Chiaki.

1.1.2 Features

- API to programatically control host and expose live audio/video stream
- Registering client for Remote Play on the host
- Interface for controlling the host, which emulates a DualShock controller
- Ability to power off/on the host if standby is enabled
- GUI which displays the live stream and supports keyboard/mouse input
- Support for controllers

1.1.3 Requirements

- Python 3.8+
- OS: Linux, Windows 10
- Note: Untested on MacOS

1.1.4 Network Requirements

This project will only work with local devices; devices on the same local network. You may be able to connect with devices on different subnets, but this is not guaranteed.

1.1.5 GUI Dependencies

The GUI requires dependencies that may be complex to install. Below is a list of such dependencies.

- pyav (May require FFMPEG to be installed)
- PySide6

`uvloop` is supported for the GUI and will be used if installed.

1.1.6 Installation

It is recommended to install in a virtual environment.

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

From pip

To install core package run:

```
pip install pyremoteplay
```

To install with optional GUI run:

```
pip install pyremoteplay[gui]
```

From Source

To Install from source, clone this repo and navigate to the top level directory.

```
pip install -r requirements.txt
python setup.py install
```

To Install GUI dependencies run:

```
pip install -r requirements-gui.txt
```

1.1.7 Setup

There are some steps that must be completed to use this library from a user standpoint.

- Registering a PSN Account
- Linking PSN Account and client to the Remote Play Host

Configuration files are saved in the `.pyremoteplay` folder in the users home directory. Both the CLI and GUI utilize the same files.

CLI Setup

Registering and linking can be completed through the cli by following the prompts after using the below command:

```
pyremoteplay {host IP Address} --register
```

Replace `{host IP Address}` with the IP Address of the Remote Play host.

GUI Setup

Registering and linking can be performed in the options screen.

1.1.8 Usage

To run the terminal only CLI use the following command: `pyremoteplay {host IP Address}`

To run the GUI use the following command: `pyremoteplay-gui`

1.1.9 Notes

- Video decoding is performed by the CPU by default. Hardware Decoding can be enabled in the options screen in the GUI.
- You may have to install `ffmpeg` with hardware decoding enabled and then install `pyav` with the following command to allow for hardware decoding: `pip install av --no-binary av`

1.1.10 Baseline measurements

The CLI instance runs at 5-10% CPU usage with around 50Mb memory usage according to `top` on this author's machine: ODroid N2.

1.1.11 Known Issues/To Do

- Text sending functions
- Add support for HDR
- Audio stutters

CHAPTER

TWO

REGISTERING

To get started, you will need to complete the following.

- Retrieving PSN account info.
- Linking PSN account with Remote Play device.

These steps can be accomplished via the CLI or GUI, but this will cover how to programatically complete these steps.

2.1 Retrieving PSN account info

This step only has to be done once per PSN user. Once the data is saved you will not have to complete this step again.

```
from pyremoteplay import RPDevice
from pyremoteplay import oauth

# This is the url that users will need to sign in
# Must be done in a web browser
url = oauth.get_login_url()

# User should be redirected to a page that says 'redirect'
# Have the user supply the url of this page
account = oauth.get_account_info(redirect_url)

# Format Account to User Profile
user_profile = oauth.format_user_account(account)

# User Profile should be saved for future use
profiles = RPDevice.get_profiles()
profiles.update_user(user_profile)
profiles.save()
```

Alternatively, you can also use the helper method *pyremoteplay.profile.Profiles.new_user()*

```
profiles.new_user(redirect_url, save=True)
```

2.2 Linking PSN account with Remote Play device

Now that we have a user profile. We can link the User to a Remote Play device. Linking needs to be performed once for each device per user.

The PSN User must be logged in on the device. The user should supply the linking PIN from 'Remote Play' settings on the device. The pin must be a string.

```
ip_address = '192.169.0.2'
device = RPDevice(ip_address)
device.get_status()  # Device needs a valid status

device.register(user_profile.name, pin, save=True)
```

CHAPTER

THREE

DEVICES

The *RPDevice* class represents a Remote Play host / console.

Ideally, most interactions should be made using this class.

Devices are identified uniquely via it's MAC address, which will differ depending on the network interface it is using (WiFi/Ethernet).

The instance will need a valid status to be usable. This can be done with the *RPDevice.get_status()* method.

Once the device has a valid status, actions can be performed such as connecting to a session, turning off/on the device.

3.1 Discovery

Devices can be discovered using the *RPDevice.search()* method. All devices that are discovered on the local network will be returned.

3.2 Creating Devices

Alternatively devices can be created manually. To create a device, the ip address or hostname needs to be known.

```
from pyremoteplay import RPDevice

device = RPDevice("192.168.86.2")
device2 = RPDevice("my_device_hostname")
```

This will create a device if the hostname is valid. However, this does not mean that the device associated with the hostname is in fact a Remote Play device.

CHAPTER

FOUR

AUDIO / VIDEO STREAM

The live audio/video stream is exposed through the *AVReceiver* class.

The *AVReceiver* class **must** be **subclassed** and have implementations for the *AVReceiver.handle_video()* and *AVReceiver.handle_audio()* methods. The audio and video frames that are passed to these methods are pyav frames.

A generic receiver is provided in this library with the *QueueReceiver* class.

4.1 Usage

To use a receiver, the receiver must be passed as a keyword argument to the *RPDevice.create_session()* method like in the example below.

```
from pyremoteplay import RPDevice
from pyremoteplay.receiver import QueueReceiver

ip_address = "192.168.86.2"
device = RPDevice(ip_address)
device.get_status()
user = device.get_users()[0]
receiver = QueueReceiver()
device.create_session(user, receiver=receiver)
```

CHAPTER

FIVE

SESSIONS

The *Session* class is responsible for connecting to a Remote Play session.

It is recommended to create a *Session* using the *RPDevice.create_session()*, method instead of creating it directly.

A *RPDevice* instance can only have one *Session* instance coupled to it at a time.

There are multiple parameters for creating a session which will configure options such as frame rate and the resolution of the video stream.

5.1 Creating a Session

The following are parameters for *RPDevice.create_session()*

The only required argument is *user*. The remaining arguments should be passed as **keyword arguments**.

Table 1: Parameters for `RPDevice.create_session()`

Parameter	Type	Default	Description
user	`str`	**<required>**	The username / PSN ID to connect with. A list of users can be found with *RPDevice.get_users()*.
profiles	*Profiles*	*None*	A profiles object. Generally not needed as the *RPDevice* class will pass this to *Session*.
loop	`asyncio.AbstractEventLoop`	*None*	The *asyncio* Event Loop to use. Must be running. Generally not needed. If not specified, the current running loop will be used.
receiver	*AVReceiver*	*None*	The receiver to use. **Note:** Must be a sub-class of AVReceiver; See *QueueReceiver*. The receiver exposes audio and video frames from the live stream. If not provided then no video/audio will be processed.
resolution	*Resolution* or `str` or `int`	*720p*	The resolution to use for video stream. Must be one of ["360p", "540p", "720p", "1080p"].
fps	*FPS* or `str` or `int`	*low*	The FPS / frame rate for the video stream. Can be expressed as ["low", "high"] or [30, 60].
quality	*Quality* or `str` or `int`	*default*	The quality of the video stream. Represents the bitrate of the stream. Must be a valid member of the *Quality* enum. Using *DEFAULT* will use the appropriate bitrate for a specific resolution.
codec	`str`	*h264*	The *FFMPEG* video codec to use. Valid codecs start with either "h264" or "hevc". There are several FFMPEG Hardware Decoding codecs that can be used such as "h264_cuvid". On devices which do not support "hevc", "h264" will always be used.

5.2 Connecting to a Session

To connect to a created session, use the async coroutine *RPDevice.connect()*.

After connecting, one should wait for it to be ready before using it. This can be done with the *RPDevice.wait_for_session()* method or the *RPDevice.async_wait_for_session()* coroutine.

The *RPDevice.ready* property will return True if the Session is ready.

5.3 Disconnecting from a Session

To disconnect, simply call the *RPDevice.disconnect()* method.

Note: This will also destroy the Session object and the *RPDevice.session* property will be set to *None*.

CHAPTER

SIX

EXAMPLES

6.1 Client

```
"""Example of running client.

We are assuming that we have already linked a PSN profile to our Remote Play device.
"""

import asyncio
import threading
import atexit

from pyremoteplay import RPDevice
from pyremoteplay.receiver import QueueReceiver


def stop(device, thread):
    loop = device.session.loop
    device.disconnect()
    loop.stop()
    thread.join(3)
    print("stopped")


def worker(device):
    loop = asyncio.new_event_loop()
    task = loop.create_task(device.connect())
    loop.run_until_complete(task)
    loop.run_forever()


def start(ip_address):
    """Return device. Start Remote Play session."""
    device = RPDevice(ip_address)
    if not device.get_status():  # Device needs a valid status to get users
        print("No Status")
        return None
    users = device.get_users()
    if not users:
        print("No users registered")
```

(continues on next page)

(continued from previous page)

```
        return None
    user = users[0]  # Gets first user name
    receiver = QueueReceiver()
    device.create_session(user, receiver=receiver)
    thread = threading.Thread(target=worker, args=(device,), daemon=True)
    thread.start()
    atexit.register(
        lambda: stop(device, thread)
    )  # Make sure we stop the thread on exit.

    # Wait for session to be ready
    device.wait_for_session()
    return device


# Usage:
#
# Starting session:
# >> ip_address = '192.168.86.2' # ip address of Remote Play device
# >> device = start(ip_address)
#
# Retrieving latest video frames:
# >> device.session.receiver.video_frames
#
# Tap Controller Button:
# >> device.controller.button("cross", "tap")
#
# Start Controller Stick Worker
# >> device.controller.start()
#
# Emulate moving Left Stick all the way right:
# >> device.controller.stick("left", axis="x", value=1.0)
#
# Release Left stick:
# >> device.controller.stick("left", axis="x", value=0)
#
# Move Left stick diagonally left and down halfway
# >> device.controller.stick("left", point=(-0.5, 0.5))
#
# Standby; Only available when session is connected:
# >> device.session.standby()
#
# Wakeup/turn on using first user:
# >> device.wakeup(device.get_users[0])
```

6.2 Async Client

```
"""Async Client Example.

This example is meant to be run as script.

We are assuming that we have already linked a PSN profile to our Remote Play device.
"""

import asyncio
import argparse

from pyremoteplay import RPDevice


async def task(device):
    """Task to run. This presses D-Pad buttons repeatedly."""
    buttons = ("LEFT", "RIGHT", "UP", "DOWN")

    # Wait for session to be ready.
    await device.async_wait_for_session()
    while device.connected:
        for button in buttons:
            await device.controller.async_button(button)
            await asyncio.sleep(1)
    print("Device disconnected")


async def get_user(device):
    """Return user."""
    if not await device.async_get_status():
        print("Could not get device status")
        return None
    users = device.get_users()
    if not users:
        print("No Users")
        return None
    user = users[0]
    return user


async def runner(host, standby):
    """Run client."""
    device = RPDevice(host)
    user = await get_user(device)
    if not user:
        return

    if standby:
        await device.standby(user)
        print("Device set to standby")
        return
```

(continues on next page)

(continued from previous page)

```
    # If device is not on, Turn On and wait for a 'On' status
    if not device.is_on:
        device.wakeup(user)
        if not await device.async_wait_for_wakeup():
            print("Timed out waiting for device to wakeup")
            return

    device.create_session(user)
    if not await device.connect():
        print("Failed to start Session")
        return

    # Now that we have connected to session we can run our task.
    asyncio.create_task(task(device))

    # This is included to keep the asyncio loop running.
    while device.connected:
        try:
            await asyncio.sleep(0)
        except KeyboardInterrupt:
            device.disconnect()
            break


def main():
    parser = argparse.ArgumentParser(description="Async Remote Play Client.")
    parser.add_argument("host", type=str, help="IP address of Remote Play host")
    parser.add_argument(
        "-s", "--standby", action="store_true", help="Place host in standby"
    )
    args = parser.parse_args()
    host = args.host
    standby = args.standby
    loop = asyncio.get_event_loop()
    loop.run_until_complete(runner(host, standby))


if __name__ == "__main__":
    main()
```

6.3 Gamepad

```
"""Example of using gamepad.

We are assuming that we have connected to a session like in 'client.py'
"""

from pyremoteplay import RPDevice
from pyremoteplay.gamepad import Gamepad
```

(continues on next page)

(continued from previous page)

```
ip_address = "192.168.0.2"
device = RPDevice(ip_address)
gamepads = Gamepad.get_all()
gamepad = gamepads[0]  # Use first gamepad

############
# After connecting to device session.
############

if not gamepad.available:
    print("Gamepad not available")
gamepad.controller = device.controller

# We can now use the gamepad.

# Load custom mapping.
gamepad.load_map("path-to-mapping.yaml")


# When done using
gamepad.close()
```

6.3.1 Mappings

For *DualShock 4* and *DualSense* controllers, the appropriate mapping will be set automatically.

Other controllers are supported but will likely need a custom mapping. This can be done by creating a *.yaml* file and then loading it at runtime.

Gamepad support is provided through pygame.

For more information on mappings see the pygame docs.

DualShock 4 Mapping Example

```
# DualShock 4 Map.

# DualShock 4 does not have hats

button:
    0: CROSS
    1: CIRCLE
    2: SQUARE
    3: TRIANGLE
    4: SHARE
    5: PS
    6: OPTIONS
    7: L3
    8: R3
    9: L1
```

(continues on next page)

(continued from previous page)

```
    10: R1
    11: UP
    12: DOWN
    13: LEFT
    14: RIGHT
    15: TOUCHPAD
axis:
    0: LEFT_X
    1: LEFT_Y
    2: RIGHT_X
    3: RIGHT_Y
    4: L2
    5: R2
hat:
```

Xbox 360 Mapping Example

```
# Xbox 360 Map

# D-Pad buttons are mapped to hat

button:
    0: CROSS
    1: CIRCLE
    2: SQUARE
    3: TRIANGLE
    4: L1
    5: R1
    6: SHARE
    7: OPTIONS
    8: L3
    9: R3
    10: PS
axis:
    0: LEFT_X
    1: LEFT_Y
    2: L2
    3: RIGHT_X
    4: RIGHT_Y
    5: R2
hat:
    left: LEFT
    right: RIGHT
    down: DOWN
    up: UP
```

CHAPTER

SEVEN

PYREMOTEPLAY

7.1 pyremoteplay package

7.1.1 Subpackages

pyremoteplay.gamepad package

Submodules

pyremoteplay.gamepad.mapping module

Mappings for Gamepad.

class pyremoteplay.gamepad.mapping.**AxisType**(*value*)

Bases: IntEnum

Axis Type Enum.

LEFT_X = 1

LEFT_Y = 2

RIGHT_X = 3

RIGHT_Y = 4

class pyremoteplay.gamepad.mapping.**HatType**(*value*)

Bases: IntEnum

Hat Type Enum.

left = 1

right = 2

down = 3

up = 4

pyremoteplay.gamepad.mapping.**rp_map_keys**()

Return RP Mapping Keys.

pyremoteplay.gamepad.mapping.**dualshock4_map**()

Return Dualshock4 Map.

Return type
dict

pyremoteplay.gamepad.mapping.**dualsense_map**()

Return DualSense Map.

Return type
dict

pyremoteplay.gamepad.mapping.**xbox360_map**()

Return XBOX 360 Map.

Return type
dict

pyremoteplay.gamepad.mapping.**default_maps**()

Return Default Maps.

Module contents

Gamepad interface to controlller.

class pyremoteplay.gamepad.**Gamepad**(*joystick*)

Bases: object

Gamepad. Wraps a PyGame Joystick to interface with RP Controller. Instances are not re-entrant after calling *close*. Creating an instance automatically starts the event loop. Instances are closed automatically when deallocated. If creating a new instance with the same joystick as an existing gamepad, the existing gamepad will be returned. This ensures that only one gamepad instance will exist per joystick.

Parameters
joystick (Union[int, Joystick]) – Either the id from *pygame.joystick.Joystick.get_instance_id()* or an instance of *pygame.joystick.Joystick*.

static joysticks()

Return All Joysticks.

static get_all()

Return All Gamepads.

static check_map(*mapping*)

Check map. Return True if valid.

Return type
bool

classmethod register(*callback*)

Register a callback with a single argument for device added/removed events.

classmethod unregister(*callback*)

Unregister a callback from receiving device added/removed events.

classmethod start()

Start Gamepad loop. Called automatically when an instance is created.

classmethod stop()

Stop Gamepad loop.

classmethod running()

Return True if running.

Return type
bool

load_map(*filename*)

Load map from file and apply. Mapping must be in *yaml* or *json* format.

Parameters
filename (str) – Absolute Path to File.

save_map(*filename*)

Save current map to file.

Parameters
filename (str) – Absolute Path to File.

default_map()

Return Default Map.

Return type
dict

close()

Close. Quit handling events.

get_hat(*hat*)

Get Hat.

Return type
HatType

get_button(*button*)

Return button value.

Return type
int

get_axis(*axis*)

Return axis value.

Return type
float

get_config()

Return Joystick config.

property controller: *Controller*

Return Controller.

Return type
Controller

property deadzone: float

Return axis deadzone. Will be positive. This represents the minimum threshold of the axis. If the absolute value of the axis is less than this value, then the value of the axis will be 0.0.

Return type
float

property mapping: dict

Return mapping.

Return type
dict

property instance_id: int

Return instance id.

Return type
int

property guid: str

Return GUID.

Return type
str

property name: str

Return Name.

Return type
str

property available: bool

Return True if available.

Return type
bool

pyremoteplay.receiver package

Module contents

AV Receivers for pyremoteplay.

class pyremoteplay.receiver.**AVReceiver**

Bases: ABC

Base Class for AV Receiver. Abstract. Must use subclass for session.

This class exposes the audio/video stream of the Remote Play Session. The *handle_video* and *handle_audio* methods need to be reimplemented. Re-implementing this class provides custom handling of audio and video frames.

AV_CODEC_OPTIONS_H264 = {'preset': 'ultrafast', 'tune': 'zerolatency'}

AV_CODEC_OPTIONS_HEVC = {'preset': 'ultrafast', 'tune': 'zerolatency'}

static audio_frame(*buf*, *codec_ctx*, *resampler=None*)

Return decoded audio frame.

Return type
AudioFrame

static video_frame(*buf*, *codec_ctx*, *video_format='rgb24'*)

Decode H264 Frame to raw image. Return AV Frame.

Parameters

- **buf** (bytes) – Raw Video Packet representing one video frame
- **codec_ctx** (CodecContext) – av codec context for decoding
- **video_format** – Format to output frames as.

Return type
VideoFrame

static video_codec(*codec_name*)

Return Video Codec Context.

Return type
CodecContext

static audio_codec(*codec_name='opus'*)

Return Audio Codec Context.

Return type
CodecContext

static audio_resampler(*audio_format='s16'*, *channels=2*, *rate=48000*)

Return Audio Resampler.

Return type
AudioResampler

decode_video_frame(*buf*)

Return decoded Video Frame.

Return type
VideoFrame

decode_audio_frame(*buf*)

Return decoded Audio Frame.

Return type
AudioFrame

handle_video_data(*buf*)

Handle video data.

handle_audio_data(*buf*)

Handle audio data.

handle_video(*frame*)

Handle video frame. Re-implementation required.

This method is called as soon as a video frame is decoded. This method should define what should happen when this frame is received. For example the frame can be stored, sent somewhere, processed further, etc.

handle_audio(*frame*)

Handle audio frame. Re-implementation required.

This method is called as soon as an audio frame is decoded. This method should define what should happen when this frame is received. For example the frame can be stored, sent somewhere, processed further, etc.

get_video_frame()

Return Video Frame. Re-implementation optional.

This method is a placeholder for retrieving a frame from a collection.

Return type
VideoFrame

get_audio_frame()

Return Audio Frame. Re-implementation optional.

This method is a placeholder for retrieving a frame from a collection.

Return type
AudioFrame

close()

Close Receiver.

property video_format

Return Video Format Name.

property video_decoder: CodecContext

Return Video Codec Context.

Return type
CodecContext

property audio_decoder: CodecContext

Return Audio Codec Context.

Return type
CodecContext

property audio_config: dict

Return Audio config.

Return type
dict

class pyremoteplay.receiver.**QueueReceiver**(*max_frames=10*, *max_video_frames=-1*, *max_audio_frames=-1*)

Bases: *AVReceiver*

Receiver which stores decoded frames in queues. New Frames are added to the end of queue. When queue is full the oldest frame is removed.

Parameters

- **max_frames** – Maximum number of frames to be stored. Will be at least 1.
- **max_video_frames** – Maximum video frames that can be stored. If <= 0, max_frames will be used.
- **max_audio_frames** – Maximum audio frames that can be stored. If <= 0, max_frames will be used.

close()

Close Receiver.

get_video_frame()

Return oldest Video Frame from queue.

Return type
VideoFrame

get_audio_frame()

Return oldest Audio Frame from queue.

Return type
AudioFrame

get_latest_video_frame()

Return latest Video Frame from queue.

Return type
VideoFrame

get_latest_audio_frame()

Return latest Audio Frame from queue.

Return type
AudioFrame

handle_video(*frame*)

Handle video frame. Add to queue.

handle_audio(*frame*)

Handle Audio Frame. Add to queue.

property video_frames: list[av.VideoFrame]

Return Latest Video Frames.

property audio_frames: list[av.AudioFrame]

Return Latest Audio Frames.

7.1.2 Submodules

pyremoteplay.const module

Constants for pyremoteplay.

class pyremoteplay.const.**StreamType**(*value*)

Bases: IntEnum

Enums for Stream type. Represents Video stream type.

Do Not Change.

H264 = 1

HEVC = 2

HEVC_HDR = 3

static parse(*value*)

Return Enum from enum, name or value.

Return type
StreamType

static preset(*value*)

Return Stream Type name.

Return type
str

class pyremoteplay.const.**Quality**(*value*)

Bases: IntEnum

Enums for quality. Value represents video bitrate.

Using *DEFAULT* will automatically find the appropriate bitrate for a specific resolution.

DEFAULT = 0

VERY_LOW = 2000

LOW = 4000

MEDIUM = 6000

HIGH = 10000

VERY_HIGH = 15000

static parse(*value*)

Return Enum from enum, name or value.

Return type
Quality

static preset(*value*)

Return Quality Value.

Return type
int

class pyremoteplay.const.**FPS**(*value*)

Bases: IntEnum

Enum for FPS.

LOW = 30

HIGH = 60

static parse(*value*)

Return Enum from enum, name or value.

Return type
FPS

static preset(*value*)

Return FPS Value.

Return type
int

class pyremoteplay.const.**Resolution**(*value*)

Bases: IntEnum

Enum for resolution.

RESOLUTION_360P = 1

RESOLUTION_540P = 2

RESOLUTION_720P = 3

RESOLUTION_1080P = 4

static parse(*value*)

Return Enum from enum, name or value.

Return type
Resolution

static preset(*value*)

Return Resolution preset dict.

Return type
dict

pyremoteplay.controller module

Controller methods.

class pyremoteplay.controller.**Controller**(*session=None*)

Bases: object

Controller Interface. Sends user input to Remote Play Session.

class ButtonAction(*value*)

Bases: IntEnum

Button Action Types.

PRESS = 1

RELEASE = 2

TAP = 3

MAX_EVENTS = 5

STATE_INTERVAL_MAX_MS = 0.2

STATE_INTERVAL_MIN_MS = 0.1

static buttons()

Return list of valid buttons.

Return type
list

connect(*session*)

Connect controller to session.

start()

Start Controller.

This starts the controller worker which listens for when the sticks move and sends the state to the host. If this is not called, the *update_sticks()* method needs to be called for the host to receive the state.

stop()

Stop Controller.

disconnect()

Stop and Disconnect Controller. Must be called to change session.

update_sticks()

Send controller stick state to host.

Will be called automatically if controller has been started with *start()*.

button(*name*, *action='tap'*, *delay=0.1*)

Emulate pressing or releasing button.

If action is *tap* this method will block by delay.

Parameters

- **name** (*Union[str, FeedbackEvent.Type]*) – The name of button. Use buttons() to show valid buttons.
- **action** (*Union[str,* ButtonAction*]*) – One of *press*, *release*, *tap*, or *Controller.ButtonAction*.
- **delay** – Delay between press and release. Only used when action is *tap*.

async async_button(*name*, *action='tap'*, *delay=0.1*)

Emulate pressing or releasing button. Async.

If action is *tap* this coroutine will sleep by delay.

Parameters

- **name** (*Union[str, FeedbackEvent.Type]*) – The name of button. Use buttons() to show valid buttons.
- **action** (*Union[str,* ButtonAction*]*) – One of *press*, *release*, *tap*, or *Controller.ButtonAction*.
- **delay** – Delay between press and release. Only used when action is *tap*.

stick(*stick_name*, *axis=None*, *value=None*, *point=None*)

Set Stick State.

If controller has not been started with *start()*, the *update_sticks()* method needs to be called manually to send stick state.

The value param represents how far to push the stick away from center.

The direction mapping is shown below:

X Axis: Left -1.0, Right 1.0

Y Axis: Up -1.0, Down 1.0

Center 0.0

Parameters

- **stick_name** – The stick to move. One of 'left' or 'right'
- **axis** – The axis to move. One of 'x' or 'y'
- **value** – The value to move stick to. Must be between -1.0 and 1.0
- **point** – An iterable of two floats, which represent coordinates. Point takes precedence over axis and value. The first value represents the x axis and the second represents the y axis

property stick_state: ControllerState

Return stick state.

Return type
ControllerState

property running: bool

Return True if running.

Return type
bool

property ready: bool

Return True if controller can be used

Return type
bool

property session: *Session*

Return Session.

Return type
Session

pyremoteplay.ddp module

Device Discovery Protocol for RP Hosts.

This module contains lower-level functions which don't need to be called directly.

pyremoteplay.ddp.**get_host_type**(*response*)

Return host type.

Parameters
response (dict) – Response dict from host

Return type
str

pyremoteplay.ddp.**get_ddp_message**(*msg_type*, *data=None*)

Get DDP message.

Parameters

- **msg_type** (str) – Message Type
- **data** (Optional[dict]) – Extra data to add

pyremoteplay.ddp.**parse_ddp_response**(*response*, *remote_address*)

Parse the response.

Parameters

- **response** (Union[str, bytes]) – Raw response from host
- **remote_address** (str) – Remote address of host

pyremoteplay.ddp.**get_ddp_search_message**()

Get DDP search message.

Return type
str

pyremoteplay.ddp.**get_ddp_wake_message**(*credential*)

Get DDP wake message.

Parameters
credential (str) – User Credential from User Profile

Return type
str

pyremoteplay.ddp.**get_ddp_launch_message**(*credential*)

Get DDP launch message.

Parameters
credential (str) – User Credential from User Profile

Return type
str

pyremoteplay.ddp.**get_socket**(*local_address='0.0.0.0'*, *local_port=9303*)

Return DDP socket.

Parameters

- **local_address** (Optional[str]) – Local address to use
- **local_port** (Optional[int]) – Local port to use

Return type
socket

pyremoteplay.ddp.**get_sockets**(*local_port=0*, *directed=None*)

Return list of sockets needed.

Parameters

- **local_port** – Local port to use
- **directed** – If True will use directed broadcast with all local interfaces.

pyremoteplay.ddp.**search**(*host='255.255.255.255'*, *local_port=9303*, *host_type=''*, *sock=None*, *timeout=3*, *directed=None*)

Return list of statuses for discovered devices.

Parameters

- **host** – Remote host to send message to. Defaults to *255.255.255.255*.

- **local_port** – Local port to use. Defaults to any.
- **host_type** – Host type. Specific host type to search for.
- **sock** – Socket. Socket will not be closed if specified.
- **timeout** – Timeout in seconds.
- **directed** – If True will use directed broadcast with all local interfaces. Sock will be ignored.

pyremoteplay.ddp.**get_status**(*host*, *local_port=9303*, *host_type=''*, *sock=None*)

Return host status dict.

Parameters

- **host** (str) – Host address
- **local_port** (int) – Local port to use
- **host_type** (str) – Host type to use
- **sock** (Optional[socket]) – Socket to use

Return type
dict

pyremoteplay.ddp.**wakeup**(*host*, *credential*, *local_port=9303*, *host_type='PS4'*, *sock=None*)

Wakeup Host.

Parameters

- **host** (str) – Host address
- **credential** (str) – User Credential from User Profile
- **local_port** (int) – Local port to use
- **host_type** (str) – Host type to use
- **sock** (Optional[socket]) – Socket to use

pyremoteplay.ddp.**launch**(*host*, *credential*, *local_port=9303*, *host_type='PS4'*, *sock=None*)

Send Launch message.

Parameters

- **host** (str) – Host address
- **credential** (str) – User Credential from User Profile
- **local_port** (int) – Local port to use
- **host_type** (str) – Host type to use
- **sock** (Optional[socket]) – Socket to use

async pyremoteplay.ddp.**async_get_socket**(*local_address='0.0.0.0'*, *local_port=0*)

Return async socket.

Parameters

- **local_address** (str) – Local address to use
- **local_port** (int) – Local port to use

Return type
AsyncUDPSocket

async pyremoteplay.ddp.**async_get_sockets**(*local_port=0*, *directed=False*)

Return list of sockets needed.

Parameters

- **local_port** – Local port to use
- **directed** – If True will use directed broadcast with all local interfaces.

pyremoteplay.ddp.**async_send_msg**(*sock*, *host*, *msg*, *host_type=''*, *directed=False*)

Send a ddp message using async socket.

Parameters

- **sock** (*AsyncUDPSocket*) – Socket to use.
- **host** (str) – Remote host to send message to.
- **msg** (str) – Message to send.
- **host_type** (str) – Host type.
- **directed** (bool) – If True will use directed broadcast with all local interfaces

async pyremoteplay.ddp.**async_search**(*host='255.255.255.255'*, *local_port=9303*, *host_type=''*, *sock=None*, *timeout=3*, *directed=None*)

Return list of statuses for discovered devices.

Parameters

- **host** – Remote host to send message to. Defaults to *255.255.255.255*.
- **local_port** – Local port to use. Defaults to any.
- **host_type** – Host type. Specific host type to search for.
- **sock** – Socket. Socket will not be closed if specified.
- **timeout** – Timeout in seconds.
- **directed** – If True will use directed broadcast with all local interfaces. Sock will be ignored.

async pyremoteplay.ddp.**async_get_status**(*host*, *local_port=9303*, *host_type=''*, *sock=None*)

Return host status dict. Async.

Parameters

- **host** (str) – Host address
- **local_port** (int) – Local port to use
- **host_type** (str) – Host type to use
- **sock** (Optional[*AsyncUDPSocket*]) – Socket to use

Return type

dict

pyremoteplay.device module

Remote Play Devices.

class pyremoteplay.device.**RPDevice**(*host*)

Bases: object

Represents a Remote Play device/host.

Most, if not all user interactions should be performed with this class. Status must be polled manually with *get_status*. Most interactions cannot be used unless there is a valid status.

Parameters
host (str) – IP address of Remote Play Host

static get_all_users(*profiles=None*)

Return all usernames that have been authenticated with OAuth.

static get_profiles(*path=''*)

Return Profiles.

Parameters
path (str) – Path to file to load profiles. If not given, will load profiles from default path.

Return type
Profiles

static search()

Return all devices that are discovered.

async static async_search()

Return all devices that are discovered. Async.

WAKEUP_TIMEOUT = 60.0

get_users(*profiles=None*)

Return Registered Users.

get_profile(*user*, *profiles=None*)

Return valid profile for user.

See: *pyremoteplay.oauth.get_user_account()* *pyremoteplay.profile.format_user_account()*

Parameters

- **user** (str) – Username of user
- **profiles** (Optional[*Profiles*]) – dict of all user profiles. If None, profiles will be retrieved from default location. Optional.

Return type
UserProfile

get_status()

Return status.

Return type
dict

async async_get_status()

Return status. Async.

set_unreachable(*state*)

Set unreachable attribute.

set_callback(*callback*)

Set callback for status changes.

create_session(*user*, *profiles=None*, *loop=None*, *receiver=None*, *resolution='720p'*, *fps='low'*, *quality='default'*, *codec='h264'*, *hdr=False*)

Return initialized session if session created else return None. Also connects a controller to session.

See *Session* for param details.

Parameters
user (str) – Name of user to use. Can be found with *get_users*

Return type
Optional[*Session*]

async connect()

Connect and start session. Return True if successful.

Return type
bool

disconnect()

Disconnect and stop session. This also sets session to None.

async standby(*user=''*, *profiles=None*)

Place Device in standby. Return True if successful.

If there is a valid and connected session, no arguments need to be passed. Otherwise creates and connects a session first.

If already connected, the sync method *RPDevice.session.standby()* is available.

Parameters
user – Name of user to use. Can be found with *get_users*

Return type
bool

wakeup(*user=''*, *profiles=None*, *key=''*)

Send Wakeup.

Either one of key or user needs to be specified. Key takes precedence over user.

Parameters

- **user** (str) – Name of user to use. Can be found with *get_users*
- **key** (str) – Regist key from registering

wait_for_wakeup(*timeout=60.0*)

Wait for device to wakeup. Blocks until device is on or for timeout.

Parameters
timeout (float) – Timeout in seconds

Return type
bool

async async_wait_for_wakeup(*timeout=60.0*)

Wait for device to wakeup. Wait until device is on or for timeout.

Parameters
timeout (float) – Timeout in seconds

Return type
bool

wait_for_session(*timeout=5*)

Wait for session to be ready. Return True if session becomes ready.

Blocks until timeout exceeded or when session is ready.

Parameters
timeout (Union[float, int]) – Timeout in seconds.

Return type
bool

async async_wait_for_session(*timeout=5*)

Wait for session to be ready. Return True if session becomes ready.

Waits until timeout exceeded or when session is ready.

Parameters
timeout (Union[float, int]) – Timeout in seconds.

Return type
bool

register(*user*, *pin*, *timeout=2.0*, *profiles=None*, *save=True*)

Register psn_id with device. Return updated user profile.

Parameters

- **user** (str) – User name. Can be found with *get_all_users*
- **pin** (str) – PIN for linking found on Remote Play Host
- **timeout** (float) – Timeout to wait for completion
- **profiles** (Optional[*Profiles*]) – Profiles to use
- **save** (bool) – Save profiles if True

Return type
UserProfile

async_register(*user*, *pin*, *timeout=2.0*, *profiles=None*, *save=True*)

Register psn_id with device. Return updated user profile.

Parameters

- **user** (str) – User name. Can be found with *get_all_users*
- **pin** (str) – PIN for linking found on Remote Play Host
- **timeout** (float) – Timeout to wait for completion
- **profiles** (Optional[*Profiles*]) – Profiles to use
- **save** (bool) – Save profiles if True

Return type
UserProfile

property host: str

Return host address.

Return type
str

property host_type: str

Return Host Type.

Return type
str

property host_name: str

Return Host Name.

Return type
str

property mac_address: str

Return Mac Address

Return type
str

property ip_address: str

Return IP Address.

Return type
str

property ddp_version: str

Return DDP Version.

Return type
str

property system_version: str

Return System Version.

Return type
str

property remote_port: int

Return DDP port of device.

Return type
int

property max_polls: int

Return max polls.

Return type
int

property unreachable: bool

Return True if unreachable

Return type
bool

property callback: Callable

Return callback for status updates.

Return type
Callable

property status: dict

Return Status as dict.

Return type
dict

property status_code: int

Return status code.

Return type
int

property status_name: str

Return status name.

Return type
str

property is_on: bool

Return True if device is on.

Return type
bool

property app_name: str

Return App name.

Return type
str

property app_id: str

Return App ID.

Return type
str

property media_info: ResultItem

Return media info.

Return type
ResultItem

property image: bytes

Return raw media image.

Return type
bytes

property session: *Session*

Return Session.

Return type
Session

property connected: bool

Return True if session connected.

Return type
bool

property ready: bool

Return True if session is ready.

Return type
bool

property controller: *Controller*

Return Controller.

Return type
Controller

pyremoteplay.errors module

Errors for pyremoteplay.

class pyremoteplay.errors.**RPErrorHandler**

Bases: object

Remote Play Errors.

class Type(*value*)

Bases: IntEnum

Enum for errors.

REGIST_FAILED = 2148567817

INVALID_PSN_ID = 2148567810

RP_IN_USE = 2148567824

CRASH = 2148567829

RP_VERSION_MISMATCH = 2148567825

UNKNOWN = 2148568063

class Message(*value*)

Bases: Enum

Messages for Error.

REGIST_FAILED = 'Registering Failed'

INVALID_PSN_ID = 'PSN ID does not exist on host'

RP_IN_USE = 'Another Remote Play session is connected to host'

CRASH = 'RP Crashed on Host; Host needs restart'

RP_VERSION_MISMATCH = 'Remote Play versions do not match on host and client'

UNKNOWN = 'Unknown'

exception pyremoteplay.errors.**RemotePlayError**

Bases: Exception

General Remote Play Exception.

exception pyremoteplay.errors.**CryptError**

Bases: Exception

General Crypt Exception.

pyremoteplay.oauth module

OAuth methods for getting PSN credentials.

pyremoteplay.oauth.**get_login_url**()

Return Login Url.

Return type
str

pyremoteplay.oauth.**get_user_account**(*redirect_url*)

Return user account.

Account should be formatted with *format_user_account()* before use.

Parameters
redirect_url (str) – Redirect url found after logging in

Return type
dict

async pyremoteplay.oauth.**async_get_user_account**(*redirect_url*)

Return user account. Async.

Account should be formatted with *format_user_account()* before use.

Parameters
redirect_url (str) – Redirect url found after logging in

Return type
dict

pyremoteplay.oauth.**prompt**()

Prompt for input and return account info.

Return type
dict

pyremoteplay.profile module

Collections for User Profiles.

These classes shouldn't be created manually. Use the helper methods such as: *pyremoteplay.profile.Profiles.load()* and *pyremoteplay.device.RPDevice.get_profiles()*

pyremoteplay.profile.**format_user_account**(*user_data*)

Format account data to user profile. Return user profile.

Parameters
user_data (dict) – User data. See *pyremoteplay.oauth.get_user_account()*

Return type
UserProfile

class pyremoteplay.profile.**HostProfile**(*name*, *data*)

Bases: UserDict

Host Profile for User.

property **name: str**

Return Name / Mac Address.

Return type
str

property **type: str**

Return type.

Return type
str

property **regist_key: str**

Return Regist Key.

Return type
str

property **rp_key: str**

Return RP Key.

Return type
str

class pyremoteplay.profile.**UserProfile**(*name*, *data*)

Bases: UserDict

PSN User Profile. Stores Host Profiles for user.

update_host(*host_profile*)

Update host profile.

Param
host_profile: Host Profile

add_regist_data(*host_status*, *data*)

Add regist data to user profile.

Parameters

- **host_status** (dict) – Status from device. See *pyremoteplay.device.RPDevice.get_status()*
- **data** (dict) – Data from registering. See *pyremoteplay.register.register()*

property **name: str**

Return PSN Username.

Return type
str

property **id: str**

Return Base64 encoded User ID.

Return type
str

property hosts: list[*HostProfile*]

Return Host profiles.

class pyremoteplay.profile.**Profiles**(*dict=None*, */*, ***kwargs*)

Bases: UserDict

Collection of User Profiles.

classmethod set_default_path(*path*)

Set default path for loading and saving.

Parameters
path (str) – Path to file.

classmethod default_path()

Return default path.

Return type
str

classmethod load(*path=''*)

Load profiles from file.

Parameters
path (str) – Path to file. If not given will use *default_path()*. File will be created automatically if it does not exist.

Return type
Profiles

new_user(*redirect_url*, *save=True*)

Create New PSN user.

See *pyremoteplay.oauth.get_login_url()*.

Parameters

- **redirect_url** (str) – URL from signing in with PSN account at the login url
- **save** – Save profiles to file if True

Return type
UserProfile

update_user(*user_profile*)

Update stored User Profile.

Parameters
user_profile (*UserProfile*) – User Profile

update_host(*user_profile*, *host_profile*)

Update host in User Profile.

Parameters

- **user_profile** (*UserProfile*) – User Profile
- **host_profile** (*HostProfile*) – Host Profile

remove_user(*user*)

Remove user.

Parameters
user (Union[str, *UserProfile*]) – User profile or user name to remove

save(*path=''*)

Save profiles to file.

Parameters
path (str) – Path to file. If not given will use default path.

get_users(*device_id*)

Return all users that are registered with a device.

Parameters
device_id – Device ID / Device Mac Address

get_user_profile(*user*)

Return User Profile for user.

Parameters
user (str) – PSN ID / Username

Return type
UserProfile

property usernames: list[str]

Return list of user names.

property users: list[*UserProfile*]

Return User Profiles.

pyremoteplay.register module

Register methods for pyremoteplay.

pyremoteplay.register.**register**(*host*, *psn_id*, *pin*, *timeout=2.0*)

Return Register info. Register this client and a PSN Account with a Remote Play Device.

Parameters

- **host** (str) – IP Address of Remote Play Device
- **psn_id** (str) – Base64 encoded PSN ID from completing OAuth login
- **pin** (str) – PIN for linking found on Remote Play Host
- **timeout** (float) – Timeout to wait for completion

Return type
dict

async pyremoteplay.register.**async_register**(*host*, *psn_id*, *pin*, *timeout=2.0*)

Return Register info. Register this client and a PSN Account with a Remote Play Device.

Parameters

- **host** (str) – IP Address of Remote Play Device
- **psn_id** (str) – Base64 encoded PSN ID from completing OAuth login

- **pin** (str) – PIN for linking found on Remote Play Host
- **timeout** (float) – Timeout to wait for completion

Return type
dict

pyremoteplay.session module

Remote Play Session.

class pyremoteplay.session.**Session**(*host*, *profile*, *loop=None*, *receiver=None*, *resolution='720p'*, *fps='low'*, *quality='default'*, *codec='h264'*, *hdr=False*)

Bases: object

Remote Play Session Async.

Parameters

- **host** (str) – IP Address of Remote Play Host
- **profile** (Union[dict, *UserProfile*]) – User Profile to connect with
- **loop** (Optional[AbstractEventLoop]) – A running asyncio event loop. If None, loop will be the current running loop
- **receiver** (Optional[*AVReceiver*]) – A receiver for handling video and audio frames
- **resolution** (Union[*Resolution*, str, int]) – The resolution of video stream. Name of or value of or *Resolution* enum
- **fps** (Union[*FPS*, str, int]) – Frames per second for video stream. Name of or value of or *FPS* enum
- **quality** (Union[*Quality*, str, int]) – Quality of video stream. Name of or value of or *Quality* enum
- **codec** (str) – Name of FFMPEG video codec to use. i.e. 'h264', 'h264_cuvid'. Video codec should be 'h264' or 'hevc'. PS4 hosts will always use h264.
- **hdr** (bool) – Uses HDR if True. Has no effect if codec is 'h264'

HEADER_LENGTH = 8

class State(*value*)

Bases: IntEnum

State Enums.

INIT = 1

RUNNING = 2

READY = 3

STOP = 4

class MessageType(*value*)

Bases: IntEnum

Enum for Message Types.

LOGIN_PIN_REQUEST = 4

LOGIN_PIN_RESPONSE = 32772

LOGIN = 5

SESSION_ID = 51

HEARTBEAT_REQUEST = 254

HEARTBEAT_RESPONSE = 510

STANDBY = 80

KEYBOARD_ENABLE_TOGGLE = 32

KEYBOARD_OPEN = 33

KEYBOARD_CLOSE_REMOTE = 34

KEYBOARD_TEXT_CHANGE_REQ = 35

KEYBOARD_TEXT_CHANGE_RES = 36

KEYBOARD_CLOSE_REQ = 37

class ServerType(*value*)

Bases: IntEnum

Server Type Enum.

UNKNOWN = -1

PS4 = 0

PS4_PRO = 1

PS5 = 2

standby(*timeout=3.0*)

Set host to standby. Blocking. Return True if successful.

Parameters
timeout – Timeout in seconds

Return type
bool

async async_standby(*timeout=3.0*)

Set host to standby. Return True if successful.

Parameters
timeout – Timeout in seconds

Return type
bool

async start(*wakeup=True*, *autostart=True*)

Start Session/RP Session.

Return type
bool

stop()

Stop Session.

set_receiver(*receiver*)

Set AV Receiver. Should be set before starting session.

wait(*timeout=5*)

Wait for session to be ready. Return True if session becomes ready.

Blocks until timeout exceeded or when session is ready.

Parameters
timeout (Union[float, int]) – Timeout in seconds.

Return type
bool

async async_wait(*timeout=5*)

Wait for session to be ready. Return True if session becomes ready.

Waits until timeout exceeded or when session is ready.

Parameters
timeout (Union[float, int]) – Timeout in seconds.

Return type
bool

property host: str

Return host address.

Return type
str

property type: str

Return host type.

Return type
str

property state: *State*

Return State.

Return type
State

property is_ready: bool

Return True if ready for user interaction.

Return type
bool

property is_running: bool

Return True if running.

Return type
bool

property is_stopped: bool

Return True if stopped.

Return type
bool

property session_id: bytes

Return Session ID.

Return type
bytes

property stream: RPStream

Return Stream.

Return type
RPStream

property stop_event: Event

Return Stop Event.

Return type
Event

property resolution: *Resolution*

Return resolution.

Return type
Resolution

property quality: *Quality*

Return Quality.

Return type
Quality

property fps: *FPS*

Return FPS.

Return type
FPS

property codec: str

Return video codec.

Return type
str

property hdr: bool

Return True if HDR.

Return type
bool

property stream_type: *StreamType*

Return Stream Type.

Return type
StreamType

property server_type: *ServerType*

Return Server Type.

Return type
ServerType

property receiver: *AVReceiver*

Return AV Receiver.

Return type
AVReceiver

property events: ExecutorEventEmitter

Return Event Emitter.

Return type
ExecutorEventEmitter

property loop: AbstractEventLoop

Return loop.

Return type
AbstractEventLoop

pyremoteplay.socket module

Async UDP Sockets. Based on asyncudp (https://github.com/eerimoq/asyncudp).

class pyremoteplay.socket.**AsyncBaseProtocol**

Bases: BaseProtocol

Base Protocol. Do not use directly.

connection_made(*transport*)

Connection Made.

connection_lost(*exc*)

Connection Lost.

error_received(*exc*)

Error Received.

async recvfrom(*timeout=None*)

Return received data and addr.

async recv(*timeout=None*)

Return received data.

Return type
Optional[bytes]

sendto(*data*, *_)

Send packet.

set_callback(*callback*)

Set callback for data received.

Setting this will flush packet received packet queue. *recv()* will always return None.

Parameters
callback – callback for data received

close()

Close transport.

get_extra_info(*name*, *default=None*)

Return Extra Info.

Return type
Any

property has_callback

Return True if callback is set.

property opened: bool

Return True if opened.

Return type
bool

property closed: bool

Return True if closed.

Return type
bool

property sock: socket

Return sock.

Return type
socket

class pyremoteplay.socket.**AsyncTCPProtocol**

Bases: Protocol, *AsyncBaseProtocol*

UDP Protocol.

connection_made(*transport*)

Connection Made.

data_received(*data*)

Called when some data is received.

The argument is a bytes object.

sendto(*data*, *_)

Send packet to address.

class pyremoteplay.socket.**AsyncUDPProtocol**

Bases: DatagramProtocol, *AsyncBaseProtocol*

UDP Protocol.

connection_made(*transport*)

Connection Made.

datagram_received(*data*, *addr*)

Datagram Received.

sendto(*data*, *addr=None*)

Send packet to address.

class pyremoteplay.socket.**AsyncBaseSocket**(*protocol*, *local_addr=None*)

Bases: object

Async Base socket. Do not use directly.

async classmethod create(*local_addr=None*, *remote_addr=None*, *, *sock=None*, ***kwargs*)

Create and return Socket.

close()

Close the socket.

sendto(*data*, *addr=None*)

Send Packet

async recv(*timeout=None*)

Receive a packet.

Return type
`Optional[bytes]`

async recvfrom(*timeout=None*)

Receive a packet and address.

get_extra_info(*name*, *default=None*)

Return Extra Info.

Return type
`Any`

setsockopt(*_AsyncBaseSocket__level*, *_AsyncBaseSocket__optname*, *_AsyncBaseSocket__value*, */*)

Set Sock Opt.

set_callback(*callback*)

Set callback for data received.

Setting this will flush packet received packet queue. *recv()* will always return None.

Parameters
callback – callback for data received

property opened: bool

Return True if opened.

Return type
`bool`

property closed: bool

Return True if closed.

Return type
`bool`

property sock: socket

Return socket.

Return type
`socket`

property local_addr: tuple[str, int]

Return local address.

class pyremoteplay.socket.**AsyncTCPSocket**(*protocol*, *local_addr=None*)

Bases: *AsyncBaseSocket*

Async TCP socket.

async classmethod create(*local_addr=None*, *remote_addr=None*, *, *sock=None*, ***kwargs*)

Create and return Socket.

send(*data*)

Send Packet.

class pyremoteplay.socket.**AsyncUDPSocket**(*protocol*, *local_addr=None*)

Bases: *AsyncBaseSocket*

Async UDP socket.

async classmethod create(*local_addr=None*, *remote_addr=None*, *, *sock=None*, *reuse_port=None*, *allow_broadcast=None*, ***kwargs*)

Create and return UDP Socket.

set_broadcast(*enabled*)

Set Broadcast enabled.

pyremoteplay.tracker module

Async Device Tracker.

class pyremoteplay.tracker.**DeviceTracker**(*default_callback=None*, *max_polls=10*, *local_port=9304*, *directed=False*)

Bases: object

Async Device Tracker.

set_max_polls(*poll_count*)

Set number of unreturned polls neeeded to assume no status.

async send_msg(*device=None*, *message=''*)

Send Message.

datagram_received(*data*, *addr*)

When data is received.

close()

Close all sockets.

add_device(*host*, *callback=None*, *discovered=False*)

Add device to track.

remove_device(*host*)

Remove device from tracking.

add_callback(*host*, *callback*)

Add callback. One per host.

remove_callback(*host*)

Remove callback from list.

async run(*interval=1*)

Run polling.

shutdown()

Shutdown protocol.

stop()

Stop Polling.

start()

Start polling.

property local_port: int

Return local port.

Return type
int

property remote_ports: dict

Return remote ports.

Return type
dict

property devices: dict

Return devices that are tracked.

Return type
dict

property device_status: list

Return all device status.

Return type
list

7.1.3 Module contents

Init file for pyremoteplay.

CHAPTER

EIGHT

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

p

INDEX

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X