raspyrfm_client Documentation

Release 1.0.0

Markus Ressel

Jun 09, 2017

Contents

A python 3.4+ library that allows the generation of network codes for the RaspyRFM rc module (and other gateways too!).

CHAPTER 1

Build Status

target https://travis-ci.org/markusressel/raspyrfm-client

CHAPTER 2

How to use

Installation

```
pip install raspyrfm-client
```

Usage

For a basic example have a look at the example.py file.

If you need more info have a look at the documentation which should help.

Basic Example

Import required modules

```
from raspyrfm_client import RaspyRFMClient
from raspyrfm_client.device_implementations.controlunit.actions import Action
from raspyrfm_client.device_implementations.controlunit.controlunit_constants import␣
↪ControlUnitModel
from raspyrfm_client.device_implementations.gateway.manufacturer.gateway_constants␣
↪import GatewayModel
from raspyrfm_client.device_implementations.manufacturer_constants import Manufacturer
```

Create the `RaspyRFMClient` object

Get a client instance by calling:

```
rfm_client = RaspyRFMClient()
```

Create a `Gateway` instance

You can let the library search automatically for gateways available in LAN using:

```
gateways = rfm_client.search()
```

This will return a list of Gateways that can later be used to send signals to.

To get a quick overview of what gateway **manufacturers** and **models** are supported call:

```
rfm_client.list_supported_gateways()
```

Create a gateway instance with the specified `IP` and `Port` of your Gateway by using: .. code-block:: python

    gateway = rfm_client.get_gateway(Manufacturer.SEEGEL_SYSTEME, GatewayModel.RASPYRFM, "192.168.2.10", 9876)

or

```
gateway = rfm_client.get_gateway(Manufacturer.SEEGEL_SYSTEME, GatewayModel.RASPYRFM,
↪"192.168.2.10") # defaults to 49880 or the gateway implementations default
```

Get a `ControlUnit`

ControlUnits are the devices that receive the RC signals sent using the gateway, f.ex. a power outlet.

To get a quick overview of what ControlUnits **manufacturers** and **models** are supported call:

```
rfm_client.list_supported_controlunits()
```

which will give you an indented list of supported manufacturers and their supported models similar to this:

```
Elro
  RC3500-A IP44 DE
  AB440S
  AB440D 200W
  AB440D 300W
  AB440ID
  AB440IS
  AB440L
  AB440SC
  AB440WD
BAT
  RC AAA1000-A IP44 Outdoor
Brennenstuhl
  RCS 1000 N Comfort
  RCS 1044 N Comfort
Intertek
  Model 1919361
[...]
```

To generate codes for a device **you first have to get an instance of its implementation** like this:

```
brennenstuhl_rcs1000 = rfm_client.get_controlunit(manufacturer_constants.BRENNENSTUHL,
                                                  manufacturer_constants.RCS_1000_N_COMFORT)
```

The parameters of the `get_controlunit()` method always need to be an enum value of the specified type. You can get an enum constant by its name though using:

```
manufacturer = Manufacturer("Intertechno")
model = ControlUnitModel("IT-1500")
```

`ControlUnit` channel configuration

Before you can generate codes with your shiny new gateway and `ControlUnit` implementations you have to specify a channel configuration for your `ControlUnit`. These **configurations can be very different for every device**. The best way to know the correct way of specifying the channel configuration for a specific device is to look at the source code (yes I know...) or by trial and error (even worse). A good `ControlUnit` implementation should tell you how the configuration should look like when specifying it in a wrong way.

However all configurations are a **keyed dictionary**. So in general there are two ways of passing the channel configuration argument. One (inline):

```
device.set_channel_config(value1=1, value2=2)
```

Two (as a dictionary):

```
device.set_channel_config(**{
    'value1': 1,
    'value2': 2
})
```

Note that the **keys always need to be a** `string`. The second one is the recommended one as it will often result in a much more readable source code.

For our Brennenstuhl device it would look like this:

```
brennenstuhl_rcs1000.set_channel_config(**{
    '1': True,
    '2': True,
    '3': True,
    '4': True,
    '5': True,
    'CH': 'A'
})
```

Generate action codes

Now that you have a properly set up `ControlUnit` you can generate codes for it's supported actions by using an `Action` enum constant that you imported previously.

To get a list of supported actions for a :code:‘ControlUnit‘call:

```
brennenstuhl_rcs1000.get_supported_actions()
```

and generate a code for one of them using your `Gateway` instance:

```
code = gateway.generate_code(brennenstuhl_rcs1000, Action.ON)
```

Send the code to the `RaspyRFM` module

To send a code for your device of choice you can combine the objects in this call:

```
rfm_client.send(gateway, brennenstuhl_rcs1000, Action.ON)
```

This will generate a code specific to the passed in gateway implementation and send it to it's host address immediately after.

CHAPTER 3

Custom implementations

The `raspyrfm-client` library is designed so you can implement custom devices in a (hopefully) very easy way.

File Structure

All `ControlUnit` implementations are located in the `/device_implementations/controlunit/manufacturer/` module and implement the base class `Device` that can be found in `/device_implementations/controlunit//base.py`.

Create a new `ControlUnit`

To create a new `ControlUnit` implementation for a new manufacturer and model create a new subdirectory for your manufacturer and a python file for your model:

Implement a `ControlUnit`

Now the basic implementation of your `ControlUnit` should looks like this:

```
from raspyrfm_client.device_implementations.controlunit.actions import Action
from raspyrfm_client.device_implementations.controlunit.base import ControlUnit


class YourModel(ControlUnit):
    def __init__(self):
        from raspyrfm_client.device_implementations.manufacturer_constants import␣
↪Manufacturer
        from raspyrfm_client.device_implementations.controlunit.controlunit_constants␣
↪import ControlUnitModel
        super().__init__(Manufacturer.YourManufacturer, ControlUnitModel.YourModel)
```

```
    def get_channel_config_args(self):
        return {}

    def get_pulse_data(self, action: Action):
        return [[0, 0], [0, 0]], 0, 0

    def get_supported_actions(self) -> [str]:
        return [Action.ON]
```

Most importantly you have to call the `super().__init__` method like shown. This will ensure that your implementation is found by the `RaspyRFMClient` and you can get an instance of your device using `rfm_client.get_controlunit()` as shown before.

If your manufacturer does not exist yet **create a new enum constant** in the `manufacturer_constants.py` file and use its value in your `__init__`. **Do the same thing for your model name** in the `controlunit_constants.py` file.

You also have to implement all abstract methods from the `Device` class. Have a look at it's documentation to get a sense of what those methods are all about.

After you have implemented all methods you are good to go! Just call `rfm_client.reload_implementation_classes()` and `rfm_client.list_supported_controlunits()` to check if your implementation is listed. If everything looks good you can use your implementation like any other one.

Exclude a WIP implementation

To prevent the RaspyRFM client from importing your half baked or base class implementation just include a class field like this:

```
class YourModel(ControlUnit):
   DISABLED = True

   [...]
```

CHAPTER 4

Contributing

GitHub is for social coding: if you want to write code, I encourage contributions through pull requests from forks of this repository. Create GitHub tickets for bugs and new features and comment on the ones that you are interested in.

CHAPTER 5

License

```
raspyrfm-client by Markus Ressel
Copyright (C) 2017  Markus Ressel

This program is free software: you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation, either version 3 of the License, or
(at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program.  If not, see <http://www.gnu.org/licenses/>.
```

CHAPTER 6

Content:

API

raspyrfm_client package

Subpackages

raspyrfm_client.device package

Subpackages

raspyrfm_client.device.manufacturer package

Subpackages

raspyrfm_client.device.manufacturer.bat package

Submodules

raspyrfm_client.device.manufacturer.bat.RC3500_A_IP44_DE module

raspyrfm_client.device.manufacturer.bat.RC_AAA1000_A_IP44_Outdoor module

Module contents

raspyrfm_client.device.manufacturer.brennenstuhl package

Submodules

raspyrfm_client.device.manufacturer.brennenstuhl.RCS1000NComfort module

raspyrfm_client.device.manufacturer.brennenstuhl.RCS1044NComfort module

Module contents

raspyrfm_client.device.manufacturer.elro package

Submodules

raspyrfm_client.device.manufacturer.elro.AB440D_200W module

raspyrfm_client.device.manufacturer.elro.AB440D_300W module

raspyrfm_client.device.manufacturer.elro.AB440ID module

raspyrfm_client.device.manufacturer.elro.AB440IS module

raspyrfm_client.device.manufacturer.elro.AB440L module

raspyrfm_client.device.manufacturer.elro.AB440S module

raspyrfm_client.device.manufacturer.elro.AB440SC module

raspyrfm_client.device.manufacturer.elro.AB440WD module

Module contents

raspyrfm_client.device.manufacturer.intertechno package

Submodules

raspyrfm_client.device.manufacturer.intertechno.CMR1000 module

raspyrfm_client.device.manufacturer.intertechno.CMR1224 module

raspyrfm_client.device.manufacturer.intertechno.CMR300 module

raspyrfm_client.device.manufacturer.intertechno.CMR500 module

raspyrfm_client.device.manufacturer.intertechno.GRR300 module

raspyrfm_client.device.manufacturer.intertechno.ITR300 module

raspyrfm_client.device.manufacturer.intertechno.ITR3500 module

raspyrfm_client.device.manufacturer.intertechno.PA31000 module

raspyrfm_client.device.manufacturer.intertechno.PAR1500 module

raspyrfm_client.device.manufacturer.intertechno.YCR1000 module

Module contents

raspyrfm_client.device.manufacturer.intertek package

Submodules

raspyrfm_client.device.manufacturer.intertek.Model1919361 module

Module contents

raspyrfm_client.device.manufacturer.mumbi package

Submodules

raspyrfm_client.device.manufacturer.mumbi.MFS300 module

Module contents

raspyrfm_client.device.manufacturer.pollin_electronic package

Submodules

raspyrfm_client.device.manufacturer.pollin_electronic.Set2605 module

Module contents

raspyrfm_client.device.manufacturer.rev package

Submodules

raspyrfm_client.device.manufacturer.rev.Ritter module

raspyrfm_client.device.manufacturer.rev.Telecontrol module

Module contents

raspyrfm_client.device.manufacturer.universal package

Submodules

raspyrfm_client.device.manufacturer.universal.HX2262Compatible module

Module contents

raspyrfm_client.device.manufacturer.vivanco package

Submodules

raspyrfm_client.device.manufacturer.vivanco.FSS31000W module

raspyrfm_client.device.manufacturer.vivanco.FSS33600W module

Module contents

Submodules

raspyrfm_client.device.manufacturer.manufacturer_constants module

Module contents

Submodules

raspyrfm_client.device.actions module

raspyrfm_client.device.base module

Module contents

Submodules

raspyrfm_client.client module

Example usage of the RaspyRFMClient can be found in the example.py file

class raspyrfm_client.client.**RaspyRFMClient**
Bases: object

This class is the main interface for generating and sending signals.

get_controlunit(*manufacturer: raspyrfm_client.device_implementations.manufacturer_constants.Manufacturer*, *model: raspyrfm_client.device_implementations.controlunit.controlunit_constants.ControlUnitModel*) → raspyrfm_client.device_implementations.controlunit.base.ControlUnit
Use this method to get a device implementation intance :param manufacturer: device manufacturer :param model: device model :return: device implementation

get_gateway(*manufacturer: raspyrfm_client.device_implementations.manufacturer_constants.Manufacturer*, *model: raspyrfm_client.device_implementations.gateway.manufacturer.gateway_constants.GatewayModel*, *host: str = None*, *port: int = None*) → raspyrfm_client.device_implementations.gateway.base.Gateway

Use this method to get a gateway implementation instance :param manufacturer: gateway manufacturer :param model: gateway model :param host: gateway host address (optional) :param port: gateway port (optional) :return: gateway implementation

get_supported_controlunit_manufacturers() → [<class 'str'>]

Returns a list of supported control unit manufacturers

get_supported_controlunit_models(*manufacturer: raspyrfm_client.device_implementations.manufacturer_constant* → [<enum 'ControlUnitModel'>]

Parameters **manufacturer** – supported control unit manufacturer

Returns a list of supported control unit models for this manufacturer

get_supported_gateway_manufacturers()

Returns a list of supported gateway manufacturers

get_supported_gateway_models(*manufacturer: raspyrfm_client.device_implementations.manufacturer_constants.Man* → [<enum 'GatewayModel'>]

Parameters **manufacturer** – supported gateway manufacturer

Returns a list of supported gateway models for this gateway manufacturer

list_supported_controlunits() → None
Prints an indented list of all supported manufacturers and models

list_supported_gateways() → None
Prints an indented list of all supported manufacturers and models

reload_implementation_classes()
Dynamically reloads device implementations

search() → [<class 'raspyrfm_client.device_implementations.gateway.base.Gateway'>]
Sends a local network broadcast with a specified message. If a gateway is present it will respond to this broadcast.

If a valid response is found the properties of this client object will be updated accordingly.

Returns list of gateways

send(*gateway: raspyrfm_client.device_implementations.gateway.base.Gateway*, *device: raspyrfm_client.device_implementations.controlunit.base.ControlUnit*, *action: raspyrfm_client.device_implementations.controlunit.actions.Action*) → None
Use this method to generate codes for actions on supported device. It will generates a string that can be interpreted by the the RaspyRFM module. The string contains information about the rc signal that should be sent.

Parameters

- **gateway** – the gateway to generate the code for
- **device** – the device to generate the code for
- **action** – action to execute

Module contents

Python Module Index

r

Index