arduino-simple-rpc

Release 2.4.2

Jeroen F.J. Laros

Nov 13, 2021

CONTENTS:

This library provides a simple way to interface to Arduino functions exported with the simpleRPC protocol. The exported method definitions are communicated to the host, which is then able to generate an API interface using this library.

Features:

- User friendly API library.
- Command line interface (CLI) for method discovery and testing.
- Function and parameter names are defined on the Arduino.
- API documentation is defined on the Arduino.
- Support for disconnecting and reconnecting.
- Support for serial and ethernet devices.

Please see ReadTheDocs for the latest documentation.

CHAPTER

ONE

QUICK START

Export any function e.g., `digitalRead()` and `digitalWrite()` on the Arduino, these functions will show up as member functions of the `Interface` class instance.

First, we make an `Interface` class instance and tell it to connect to the serial device `/dev/ttyACM0`.

```
>>> from simple_rpc import Interface
>>>
>>> interface = Interface('/dev/ttyACM0')
```

We can use the built-in `help()` function to see the API documentation of any exported method.

```
>>> help(interface.digital_read)
Help on method digital_read:

digital_read(pin) method of simple_rpc.simple_rpc.Interface instance
    Read digital pin.

    :arg int pin: Pin number.

    :returns int: Pin value.
```

All exposed methods can be called like any other class method.

```
>>> interface.digital_read(8)         # Read from pin 8.
0
>>> interface.digital_write(13, True) # Turn LED on.
```

CHAPTER

TWO

FURTHER READING

For more information about the host library and other interfaces, please see the Usage and Library sections.

2.1 Introduction

This Python library provides a simple way to interface to Arduino functions exported with the simpleRPC protocol.

For more background information and the reasons that led to this project, see the motivation section of the device library documentation.

This project serves as a reference implementation for clients using the simpleRPC protocol.

2.2 Installation

The software is distributed via PyPI, it can be installed with `pip`.

```
pip install arduino-simple-rpc
```

2.2.1 From source

The source is hosted on GitHub, to install the latest development version, use the following commands.

```
git clone https://github.com/jfjlaros/arduino-simple-rpc.git
cd arduino-simple-rpc
pip install .
```

Development

Tests are written in the pytest framework which can be installed with `pip`.

```
pip install pytest
```

To run the automated tests, run `py.test` in the root of the project folder.

By default, all tests that rely on particular hardware to be connected are disabled. The `--device` parameter can be used to enable these device specific tests.

To test the Bluetooth interface.

```
py.test --device bt
```

To test the HardwareSerial interface.

```
py.test --device serial
```

To test the WiFi interface.

```
py.test --device wifi
```

2.3 Usage

The command line interface can be useful for method discovery and testing purposes. It currently has two subcommands: `list`, which shows a list of available methods and `call` for calling methods. For more information, use the `-h` option.

```
$ simple_rpc -h
```

Note: Please note that the initialisation procedure has a built in two second delay which can be modified with the `-w` parameter. For each invocation of `list` or `call`, the device is reset and reinitialised, so using the command line interface for time critical or high speed applications is not advised. For these types of applications, the *Library* should be used directly instead.

2.3.1 Connecting

To detect serial devices, we recommend using the arduino-cli toolkit.

```
$ arduino-cli board list
Port         Type              Board Name                FQBN             Core
/dev/ttyACM0 Serial Port (USB) Arduino Mega or Mega 2560 arduino:avr:mega arduino:avr
```

This command will not detect any devices connected via ethernet or WiFi. Use a URL (e.g., `socket://192.168.1.50:10000`) instead.

2.3.2 Method discovery

When the device is known, the `list` subcommand can be used to retrieve all available methods.

```
$ simple_rpc list /dev/ttyACM0
```

Alternatively, for ethernet and WiFi devices.

```
$ simple_rpc list socket://192.168.1.50:10000
```

If the Arduino has exposed the functions `inc` and `set_led` like in the example given in the device library documentation, the `list` subcommand will show the following.

```
inc a
    Increment a value.

    int a: Value.

    returns int: a + 1.


set_led brightness
    Set LED brightness.

    int brightness: Brightness.
```

2.3.3 Calling a method

Any of the methods can be called by using the `call` subcommand.

```
$ simple_rpc call /dev/ttyACM0 inc 1
2
```

Alternatively, for ethernet or WiFi devices.

```
$ simple_rpc call socket://192.168.1.50:10000 inc 1
2
```

Please see the list of handlers for a full description of the supported interface types.

2.3.4 Complex objects

Complex objects are passed on the command line interface as a JSON string. Binary encoding and decoding is taken care of by the CLI. The following example makes use of the demo sketch in the device examples.

```
$ simple_rpc call /dev/ttyACM0 vector '[1, 2, 3, 4]'
[1.40, 2.40, 3.40, 4.40]

$ simple_rpc call /dev/ttyACM0 object '["a", [10, "b"]]'
["b", [11, "c"]]
```

2.3.5 Low throughput networks

When working with low throughput networks (e.g., LoRa), device initialisation can take a long time. To counteract this problem, it is possible to save the interface definition to a file, which can subsequently be used to initialise the interface without having to query the device.

An interface definition can be saved to a file using the `-s` option of the `list` subcommand.

```
$ simple_rpc list -s interface.yml /dev/ttyACM0
```

A saved interface definition can be loaded to skip the initialisation procedure by using the `-l` option of the `call` subcommand.

```
$ simple_rpc call -l interface.yml /dev/ttyACM0 inc 1
2
```

2.4 Command Line Interface

Arduino simpleRPC API client library and CLI.

```
usage: simple_rpc [-h] [-v] {list,call} ...
```

2.4.1 Positional Arguments

subcommand Possible choices: list, call

2.4.2 Named Arguments

-v show program's version number and exit

2.4.3 Sub-commands:

list

List the device methods.

```
simple_rpc list [-h] [-o OUTPUT] [-b BAUDRATE] [-w WAIT] [-s SAVE] DEVICE
```

Positional Arguments

DEVICE device

Named Arguments

-o output file

Default: -

-b baud rate

Default: 9600

-w time before communication starts

Default: 2

-s interface definition file

call

Execute a method.

```
simple_rpc call [-h] [-o OUTPUT] [-b BAUDRATE] [-w WAIT] [-l LOAD]
                DEVICE NAME [ARG [ARG ...]]
```

Positional Arguments

DEVICE	device
NAME	command name
ARG	command parameter

Named Arguments

-o	output file
	Default: -
-b	baud rate
	Default: 9600
-w	time before communication starts
	Default: 2
-l	interface definition file

2.5 Library

The API library provides several interfaces, discussed below. All interfaces share the methods described in Section *Methods*.

2.5.1 Generic interface

The `Interface` class can be used when the type of device is not known beforehand, A class instance is made by passing either the path to a device or a URI to the constructor.

```
>>> from simple_rpc import Interface
>>> interface = Interface('/dev/ttyACM0')
```

The constructor takes the following parameters.

Table 1: Constructor parameters.

name	optional	description
`device`	no	Device name.
`baudrate`	yes	Baud rate.
`wait`	yes	Time in seconds before communication starts.
`autoconnect`	yes	Automatically connect.
`load`	yes	Load interface definition from file.

Please see the list of handlers for a full description of the supported interface types.

Serial interface

When a path to a serial device is given, the `Interface` constructor returns a `SerialInterface` class instance.

```
>>> from simple_rpc import Interface
>>> interface = Interface('/dev/ttyACM0')
>>> interface.__class__
<class 'simple_rpc.simple_rpc.SerialInterface'>
```

Alternatively, the `SerialInterface` class can be used directly.

```
>>> from simple_rpc import SerialInterface
>>> interface = SerialInterface('/dev/ttyACM0')
```

Socket interface

When a socket URI is given, the `Interface` constructor returns a `SocketInterface` class instance.

```
>>> interface = Interface('socket://192.168.1.50:10000')
>>> interface.__class__
<class 'simple_rpc.simple_rpc.SocketInterface'>
```

Alternatively, the `SocketInterface` class can be used directly.

```
>>> from simple_rpc import SocketInterface
>>> interface = SocketInterface('socket://192.168.1.50:10000')
```

Methods

The `Interface` class provides the following methods.

Table 2: Class methods.

name	description
`open()`	Connect to device.
`close()`	Disconnect from device.
`is_open()`	Query device state.
`call_method()`	Execute a method.
`save()`	Save the interface definition to a file.

The `open()` function is used to connect to a device, this is needed when `autoconnect=False` is passed to the constructor.

```
>>> interface = Interface('/dev/ttyACM0', autoconnect=False)
>>> # Do something.
>>> interface.open()
```

The `open()` function accepts the optional parameter `handle`, which can be used to load an interface definition from a file. This can be useful when working with low throughput networks.

```
>>> interface.open(open('interface.yml'))
```

The connection state can be queried using the `is_open()` function and it can be closed using the `close()` function.

```
>>> if interface.is_open():
>>>     interface.close()
```

Additionally, the `with` statement is supported for easy opening and closing.

```
>>> with Interface('/dev/ttyACM0') as interface:
>>>     interface.ping(10)
```

The class instance has a public member variable named `device` which contains the device definitions and its exported method definitions.

```
>>> list(interface.device['methods'])
['inc', 'set_led']
```

Example of a method definition.

```
>>> interface.device['methods']['inc']
{
  'doc': 'Increment a value.',
  'index': 2,
  'name': 'inc',
  'parameters': [
    {
      'doc': 'Value.',
      'name': 'a',
      'fmt': 'h',
      'typename': 'int'
    }
  ],
  'return': {
    'doc': 'a + 1.',
    'fmt': 'h',
    'typename': 'int'}
}
```

Every exported method will show up as a class method of the `interface` class instance. These methods can be used like any normal class methods. Alternatively, the exported methods can be called by name using the `call_method()` function.

The `save()` function is used to save the interface definition to a file. This can later be used by the constructor or the `open()` function to initialise the interface without having to query the device.

```
>>> interface.save(open('interface.yml', 'w'))
```

2.5.2 Basic usage

In the example given in the device library documentation, the `inc` method is exported, which is now present as a class method of the class instance.

```
>>> interface.inc(1)
2
```

Alternatively, the exported method can be called using the `call_mathod()` function.

```
>>> interface.call_method('inc', 1)
2
```

To get more information about this class method, the built-in `help()` function can be used.

```
>>> help(interface.inc)
Help on method inc:

inc(a) method of simple_rpc.simple_rpc.SerialInterface instance
    Increment a value.

    :arg int a: Value.

    :returns int: a + 1.
```

Note that strings should be encoded as `bytes` objects. If, for example, we have a function named `test` that takes a string as parameter, we should call this function as follows.

```
>>> interface.test(b'hello world')
```

2.5.3 Complex objects

Some methods may have complex objects like Tuples, Objects or Vectors as parameters or return type.

In the following example, we call a method that takes a Vector of integers and returns a Vector of floats.

```
>>> interface.vector([1, 2, 3, 4])
[1.40, 2.40, 3.40, 4.40]
```

In this example, we call a method that takes an Object containing a byte and an other Object. A similar Object is returned.

```
>>> interface.object((b'a', (10, b'b')))
(b'b', (11, b'c'))
```

Note: In this implementation, the `Tuple` type is regarded as a flat sequence of elements, not as a separate type. The `Object` type is used to assign this sequence of elements to a Python `tuple` and the `Vector` type is used to assign this sequence of elements to a Python `list`. Bare tuples are not supported in this implementation.

Using Tuples may lead to some other counter intuitive results. For example, a Vector of length l containing Tuples of size n is received as a list containing $l \cdot n$ elements.

2.6 API documentation

2.6.1 SimpleRPC

class simple_rpc.simple_rpc.**Interface**(*device: str*, *baudrate: int = 9600*, *wait: int = 2*, *autoconnect: bool = True*, *load: TextIO = None*)

Generic simpleRPC interface wrapper.

Parameters

- **device** – Device name.
- **baudrate** – Baud rate.
- **wait** – Time in seconds before communication starts.
- **autoconnect** – Automatically connect.
- **load** – Load interface definition from file.

class simple_rpc.simple_rpc.**SerialInterface**(*device*, *baudrate=9600*, *wait=2*, *autoconnect=True*, *load=None*)

Serial simpleRPC interface.

Parameters

- **device** (str) – Device name.
- **baudrate** (int) – Baud rate.
- **wait** (int) – Time in seconds before communication starts.
- **autoconnect** (bool) – Automatically connect.
- **load** (Optional[TextIO]) – Load interface definition from file.

call_method(*name*, **args*)

Execute a method.

Parameters

- **name** (str) – Method name.
- **args** (Any) – Method parameters.

Return type Any

Returns Return value of the method.

close()

Disconnect from device.

Return type None

is_open()

Query interface state.

Return type bool

open(*handle=None*)
Connect to device.

Parameters **handle** (Optional[TextIO]) – Open file handle.

Return type None

save(*handle*)
Save the interface definition to a file.

Parameters **handle** (TextIO) – Open file handle.

Return type None

class simple_rpc.simple_rpc.**SocketInterface**(*device*, *baudrate=9600*, *wait=2*, *autoconnect=True*, *load=None*)
Socket simpleRPC interface.

Parameters

- **device** (str) – Device name.
- **baudrate** (int) – Baud rate.
- **wait** (int) – Time in seconds before communication starts.
- **autoconnect** (bool) – Automatically connect.
- **load** (Optional[TextIO]) – Load interface definition from file.

call_method(*name*, **args*)
Execute a method.

Parameters

- **name** (str) – Method name.
- **args** (Any) – Method parameters.

Return type Any

Returns Return value of the method.

close()
Disconnect from device.

Return type None

is_open()
Query interface state.

Return type bool

open(*handle=None*)
Connect to device.

Parameters **handle** (Optional[TextIO]) – Open file handle.

Return type None

save(*handle*)
Save the interface definition to a file.

Parameters **handle** (TextIO) – Open file handle.

Return type None

2.6.2 Protocol

simple_rpc.protocol.**parse_line**(*index*, *line*)
Parse a method definition line.

Parameters

- **index** (int) – Line number.
- **line** (bytes) – Method definition.

Return type dict

Returns Method object.

2.6.3 Extras

simple_rpc.extras.**dict_to_object**(*d*)
Convert a dictionary using UTF-8 to an object using binary strings.

Parameters **d** (dict) – Dictionary with UTF-8 encoded strings.

Return type object

Returns Object with binary encoded strings.

simple_rpc.extras.**json_utf8_decode**(*obj*)
Decode all strings in an object to UTF-8.

Parameters **obj** (object) – Object.

Return type object

Returns Object with UTF-8 encoded strings.

simple_rpc.extras.**json_utf8_encode**(*obj*)
Binary encode all strings in an object.

Parameters **obj** (object) – Object.

Return type object

Returns Object with binary encoded strings.

simple_rpc.extras.**make_function**(*method*)
Make a member function for a method.

Parameters **method** (dict) – Method object.

Return type callable

Returns New member function.

simple_rpc.extras.**object_to_dict**(*obj*)
Convert an object using binary strings to a dictionary using UTF-8.

Parameters **obj** (object) – Object with binary encoded strings.

Return type dict

Returns Dictionary with UTF-8 encoded strings.

2.7 Contributors

- Jeroen F.J. Laros <jlaros@fixedpoint.nl> (Original author, maintainer)
- Chris Flesher <chris.flesher@stoneaerospace.com> (Ethernet support)

Find out who contributed:

```
git shortlog -s -e
```

PYTHON MODULE INDEX

S

INDEX