
SimulRPI

Release 0.1

Raul C.

Aug 14, 2020

CONTENTS

1	README	3
1.1	Introduction	4
1.2	Dependencies	4
1.3	Installation instructions	4
1.4	Usage	4
1.5	Examples	7
1.6	Change Log	11
1.7	TODOs	11
1.8	Resources	12
1.9	References	12
2	API Reference	13
2.1	<code>SimulRpi.GPIO</code>	13
2.2	<code>SimulRpi.mapping</code>	22
2.3	<code>SimulRpi.run_examples</code>	24
2.4	<code>SimulRpi.utils</code>	27
3	Indices and tables	29
	Python Module Index	31
	Index	33

SimulRPI is a Python library that partly fakes [RPi.GPIO](#) and simulates some I/O devices on a Raspberry Pi (RPI).

Each circle represents a blinking LED connected to an RPi and the number between brackets is the associated GPIO channel number. Here the “LED” on channel 22 toggles between on and off when a key is pressed.

See the [README](#) for more info about the library.

README

SimulRPI is a Python library that partly fakes `RPi.GPIO` and simulates some I/O devices on a Raspberry Pi (RPI).

- *Introduction*
- *Dependencies*
- *Installation instructions*
- *Usage*
 - *Use the library in your own code*
 - * *Case 1: with a `try` and `except` blocks*
 - * *Case 2: with a simulation flag*
 - *Script `run_examples.py`*
 - * *List of options*
 - * *How to run the script*
- *Examples*
 - *Example 1: display 1 LED*
 - *Example 2: display 3 LEDs*
 - *Example 3: detect a pressed key*
 - *Example 4: blink a LED*
 - *Example 5: blink a LED if a key is pressed*
- *Change Log*
 - *0.0.1a0*
 - *0.0.0a0*
- *TODOs*
- *Resources*
- *References*

1.1 Introduction

In addition to partly faking `RPi.GPIO`, **SimulRPI** also simulates these I/O devices connected to an RPi:

- push buttons by listening to pressed keyboard keys and
- LEDs by displaying small dots blinking on the terminal along with their GPIO pin number.

When a LED is turned on, it is shown as a small red circle on the terminal. The package `pynput` is used to monitor the keyboard for any pressed key.

Example: terminal output

Each circle represents a blinking LED connected to an RPi and the number between brackets is the associated GPIO channel number. Here the “LED” on channel 22 toggles between on and off when a key is pressed.

Important: This library is not a Raspberry Pi emulator nor a complete mock-up of `RPi.GPIO`, only the most important functions that I needed for my [Darth-Vader-RPi](#) project were added.

If there is enough interest in this library, I will eventually mock more functions from `RPi.GPIO`. Thus, [let me know through SimulRPI's issues page](#) if you want me to add more things to this library.

1.2 Dependencies

- **Platforms:** macOS, Linux
- **Python:** 3.5, 3.6, 3.7, 3.8
- `pynput >=1.6.8`: for monitoring the keyboard for any pressed key

1.3 Installation instructions

1. Install the `SimulRPI` package with `pip`:

```
$ pip install SimulRPI
```

It will install the dependency `pynput` if it is not already found in your system.

2. Test your installation by importing `SimulRPI` and printing its version:

```
$ python -c "import SimulRPI; print(SimulRPI.__version__)"
```

1.4 Usage

1.4.1 Use the library in your own code

Case 1: with a `try` and `except` blocks

You can try importing `RPi.GPIO` first and if it is not found, then fallback on the module `SimulRPI.GPIO`.

Listing 1: **Case 1:** with a `try` and `except` blocks

```
try:
    import RPi.GPIO as GPIO
except ImportError:
    import SimulRPI.GPIO as GPIO

# Rest of your code
```

The code from the previous example would be put at the beginning of your file with the other imports.

Case 2: with a simulation flag

Or maybe you have a flag to tell whether you want to work with the simulation module or the real one.

Listing 2: **Case 2:** with a simulation flag

```
if simulation:
    import SimulRPI.GPIO as GPIO
else:
    import RPi.GPIO as GPIO

# Rest of your code
```

1.4.2 Script `run_examples.py`

The script `run_examples` which you have access to once you *install* the `SimulRPI` package allows you to run different code examples on your RPi or computer. If it is run on your computer, it will make use of the module `SimulRPI.GPIO` which partly fakes `RPi.GPIO`.

The different code examples are those presented in *Examples* and show the capability of `SimulRPI.GPIO` for simulating I/O devices on an RPi such as push buttons and LEDs.

Here is a list of the functions associated with each code example:

- Example 1: `run_examples.ex1_turn_on_led()`
- Example 2: `run_examples.ex2_turn_on_many_leds()`
- Example 3: `run_examples.ex3_detect_button()`
- Example 4: `run_examples.ex4_blink_led()`
- Example 5: `run_examples.ex5_blink_led_if_button()`

List of options

To display the script's list of options and their descriptions: `run_examples -h`

-e	The number of the code example you want to run. It is required. (default: None)
-m	Set the numbering system used to identify the I/O pins on an RPi. (default: BCM)
-s	Enable simulation mode, i.e. <code>SimulRPI.GPIO</code> will be used for simulating <code>RPi.GPIO</code> . (default: False)

-l	The GPIO channels to be used for LEDs. If an example only requires 1 channel, the first channel from the provided list will be used. (default: [10, 11, 12])
-b	The GPIO channel to be used for a push button. The default value is channel 20 which is associated with the keyboard key <i>alt_r</i> . (default: 13)
-t	Total time in seconds LEDs will be blinking. (default: 4)
-k	The name of the key associated with the button channel. The name must be one of those recognized by the module <i>pynput</i> . See the <i>SimulRPI</i> documentation for a list of valid key names: https://bit.ly/2Pw10Be . Example: <i>alt</i> , <i>cmd_r</i> (default: <i>alt_r</i>)
--on	Time in seconds the LED will stay turned ON at a time. (default: 1)
--off	Time in seconds the LED will stay turned OFF at a time. (default: 1)

How to run the script

Once you install the package *SimulRPI* (see [Installation Instructions](#)), you should have access to the script `run_examples` which can be called from the terminal by providing some arguments.

For example: `run_examples -e 1 -s`.

Let's run the code example # 5 which blinks a LED if a specified key is pressed.

Here is the command line for blinking a LED (on channel 21) for a total of 5 seconds if the key `cmd_r` is pressed when the simulation package *SimulRPI* is used:

```
$ run_examples -s -e 5 -l 21 -t 5 -k cmd_r
```

Output:

Important: Don't forget the flag `-s` (for simulation) when running the script `run_examples` if you want to run a code example on your computer, and not on your RPi.

1.5 Examples

The examples presented thereafter will show you how to use `SimulRPI` to simulate LEDs and push buttons.

The code for the examples shown here can be also found as a script in `run_examples`.

Note: Since we are showing how to use the `SimulRPI` library, the presented code examples are to be executed on your computer. However, the script `run_examples.py` which runs the following code examples can be executed on a Raspberry Pi or your computer.

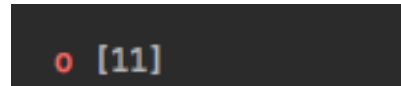
1.5.1 Example 1: display 1 LED

Example 1 consists in displaying one LED on the GPIO channel 11. Here is the code along with the output from the terminal:

```
import SimulRPI.GPIO as GPIO

led_channel = 11
GPIO.setmode(GPIO.BCM)
GPIO.setup(led_channel, GPIO.OUT)
GPIO.output(led_channel, GPIO.HIGH)
GPIO.cleanup()
```

Output:

A terminal window with a dark background. It displays a red LED symbol (a circle with a vertical line) followed by the text `[11]` in a light blue or white monospace font.

The command line for reproducing the same results for example 1 with the script `run_examples` is the following:

```
$ run_examples -s -e 1 -l 11
```

Warning: Always call `GPIO.cleanup()` at the end of your program to free up any resources such as stopping threads.

1.5.2 Example 2: display 3 LEDs

Example 2 consists in displaying three LEDs on channels 10, 11, and 12, respectively. Here is the code along with the output from the terminal:

```
import SimulRPI.GPIO as GPIO

led_channels = [10, 11, 12]
GPIO.setmode(GPIO.BCM)
for ch in led_channels:
    GPIO.setup(ch, GPIO.OUT)
    GPIO.output(ch, GPIO.HIGH)
GPIO.cleanup()
```

Output:



```
o [10]  o [11]  o [12]
```

The command line for reproducing the same results for example 2 with the script `run_examples` is the following:

```
$ run_examples -s -e 2
```

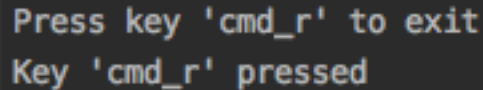
1.5.3 Example 3: detect a pressed key

Example 3 consists in detecting if the key `cmd_r` is pressed and then printing a message. Here is the code along with the output from the terminal:

```
import SimulRPI.GPIO as GPIO

channel = 17
GPIO.setmode(GPIO.BCM)
GPIO.setup(channel, GPIO.IN, pull_up_down=GPIO.PUD_UP)
print("Press key 'cmd_r' to exit")
while True:
    if not GPIO.input(channel):
        print("Key 'cmd_r' pressed")
        break
GPIO.cleanup()
```

Output:



```
Press key 'cmd_r' to exit
Key 'cmd_r' pressed
```

The command line for reproducing the same results for example 3 with the script `run_examples` is the following:

```
$ run_examples -s -e 3 -k cmd_r
```

Note: By default, SimulRPI maps the key `cmd_r` to channel 17 as can be seen from the [default key-to-channel map](#).

See also the documentation for [SimulRPI.mapping](#) where the default keymap is defined.

1.5.4 Example 4: blink a LED

Example 4 consists in blinking a LED on channel 20 for 4 seconds (or until you press `ctrl + c`). Here is the code along with the output from the terminal:

```
import time
import SimulRPI.GPIO as GPIO

channel = 20
GPIO.setmode(GPIO.BCM)
GPIO.setup(channel, GPIO.OUT)
start = time.time()
while (time.time() - start) < 4:
    try:
        GPIO.output(channel, GPIO.HIGH)
        time.sleep(0.5)
        GPIO.output(channel, GPIO.LOW)
        time.sleep(0.5)
    except KeyboardInterrupt:
        break
GPIO.cleanup()
```

Output:

The command line for reproducing the same results for example 4 with the script `run_examples` is the following:

```
$ run_examples -s -e 4 -t 4 -l 20
```

1.5.5 Example 5: blink a LED if a key is pressed

Example 5 consists in blinking a LED on channel 10 for 3 seconds if the key `ctrl_r` is pressed. And then, exiting from the program. The program can also be terminated at any time by pressing `ctrl + c`. Here is the code along with the output from the terminal:

```
import time
import SimulRPI.GPIO as GPIO

led_channel = 10
key_channel = 20
GPIO.setmode(GPIO.BCM)
GPIO.setup(led_channel, GPIO.OUT)
GPIO.setup(key_channel, GPIO.IN, pull_up_down=GPIO.PUD_UP)
print("Press key 'ctrl_r' to blink a LED")
while True:
    try:
        if not GPIO.input(key_channel):
            print("Key 'ctrl_r' pressed")
            start = time.time()
            while (time.time() - start) < 3:
                GPIO.output(led_channel, GPIO.HIGH)
                time.sleep(0.5)
                GPIO.output(led_channel, GPIO.LOW)
                time.sleep(0.5)
            break
    except KeyboardInterrupt:
        break
GPIO.cleanup()
```

Output:

The command line for reproducing the same results for example 5 with the script `run_examples` is the following:

```
$ run_examples -s -e 5 -t 3 -k ctrl_r
```

Note: By default, `SimulRPI` maps the key `ctrl_r` to channel 20 as can be from the [default key-to-channel map](#). See also the documentation for [SimulRPI.mapping](#) where the default keymap is defined.

1.6 Change Log

1.6.1 0.0.1a0

- In `SimulRPI.GPIO`, the package `pynput` is not required anymore. If it is not found, all keyboard-related functionalities from the `SimulRPI` library will be skipped. Thus, no keyboard keys will be detected if pressed or released when `pynput` is not installed.

This was necessary because *Travis* was raising an exception when I was running a unit test: `Xlib.error.DisplayNameError`. It was due to `pynput` not working well in a headless setup. Thus, `pynput` is now removed from `requirements_travis.txt`.

Eventually, I will mock `pynput` when doing unit tests on parts of the library that make use of `pynput`.

- Started writing unit tests

1.6.2 0.0.0a0

- First version
 - Tested code *examples* on different platforms and here are the results
 - On an RPi with `RPI.GPIO`: all examples involving LEDs and pressing buttons worked
 - On a computer with `SimulRPI.GPIO`
 - * macOS: all examples involving “LEDs” and keyboard keys worked
 - * RPi OS [Debian-based]: all examples involving only “LEDs” worked
- NOTE:** I was running the script *run_examples* with `ssh` but `pynput` doesn't detect any pressed keyboard keys even though I set my environment variable `Display`, added `PYTHONPATH` to *etc/sudoers* and ran the script with `sudo`. To be further investigated.

1.7 TODOs

- Run code *examples* involving pressing keyboard keys directly on an RPi (no `ssh`) and post results. **High priority**
- In *run_examples*, improve timer accuracy when waiting for a LED to stop blinking or for a function to stop displaying a LED. **Medium priority**
- Mock `pynput` when doing unit tests on Travis. **Medium priority**
- Investigate further why no keyboard keys could be detected when connecting to an RPi through `ssh` and running the script *run_examples* with `sudo`. **Low priority**

1.8 Resources

- [SimulRPI GitHub](#): source code
- [SimulRPI PyPI](#)
- [Darth-Vader-RPi](#): personal project using `RPi.GPIO` for activating a Darth Vader action figure with light and sounds and `SimulRPi.GPIO` as fallback if testing on a computer when no RPi available

1.9 References

- [pynput](#): package used for monitoring the keyboard for any pressed keys as to simulate push buttons connected to an RPi
- [RPi.GPIO](#): a module to control RPi GPIO channels

API REFERENCE

- `SimulRPi.GPIO`
- `SimulRPi.mapping`
- `SimulRPi.run_examples`
 - *Usage*
- `SimulRPi.utils`

2.1 SimulRPi.GPIO

Module that partly fakes `RPi.GPIO` and simulates some I/O devices.

It simulates these I/O devices connected to a Raspberry Pi:

- push buttons by listening to pressed keyboard keys and
- LEDs by displaying small circles blinking on the terminal along with their GPIO pin number.

When a LED is turned on, it is shown as a small red circle on the terminal. The package `pynput` is used to monitor the keyboard for any pressed key.

Example: terminal output

```
o [11]    o [9]    o [10]
```

where each circle represents a LED (here they are all turned off) and the number between brackets is the associated GPIO pin number.

Important: This library is not a Raspberry Pi emulator nor a complete mock-up of `RPi.GPIO`, only the most important functions that I needed for my [Darth-Vader-RPi project](#) were added.

If there is enough interest in this library, I will eventually mock more functions from `RPi.GPIO`. Thus, [let me know through SimulRPi's issues page](#) if you want me to add more things to this library.

class `GPIO.Manager`

Bases: `object`

Class that manages the pin database (`PinDB`) and the threads responsible for displaying “LEDs” on the terminal and listening for keys pressed/released.

The threads are not started right away in `__init__()` but in `input()` for the listener thread and `output()` for the displaying thread.

They are eventually stopped in `cleanup()`.

Variables

- **mode** (`int`) – Numbering system used to identify the I/O pins on an RPi: *BOARD* or *BCM*. Default value is *None*.
- **warnings** (`bool`) – Whether to show warnings when using a pin other than the default GPIO function (input). Default value is *True*.
- **enable_printing** (`bool`) – Whether to enable printing on the terminal. Default value is *True*.
- **pin_db** (*PinDB*) – A *Pin* database. See *PinDB* on how to access it.
- **key_to_channel_map** (`dict`) – A dictionary that maps keyboard keys (`string`) to GPIO channel numbers (`int`). By default, it takes the keys and values defined in *SimulRPI.mapping*'s keymap `default_key_to_channel_map`.
- **channel_to_key_map** (`dict`) – The reverse dictionary of `key_to_channel_map`. It maps channels to keys.
- **nb_prints** (`int`) – Number of times the displaying thread `th_display_leds` has printed blinking circles on the terminal. It is used when debugging the displaying thread.
- **th_display_leds** (`threading.Thread`) – Thread responsible for displaying small blinking circles on the terminal as to simulate LEDs connected to an RPi.
- **th_listener** (`keyboard.Listener`) – Thread responsible for listening on any pressed or released key as to simulate push buttons connected to an RPi.

Note: A keyboard listener is a `threading.Thread`, and all callbacks will be invoked from the thread.

Ref.: <https://pynput.readthedocs.io/en/latest/keyboard.html#monitoring-the-keyboard>

Important: If the module `pynput.keyboard` couldn't be imported, the listener thread `th_listener` will not be created and the parts of the *SimulRPI* library that monitors the keyboard for any pressed or released key will be ignored. Only the thread `th_display_leds` that displays “LEDs” on the terminal will be created.

This is necessary for example in the case we are running tests on travis and we don't want travis to install `pynput` in a headless setup because an exception will get raised:

```
Xlib.error.DisplayNameError: Bad display name ""
```

The tests involving `pynput` will be performed with a mock version of `pynput`.

add_pin (*channel*, *gpio_function*, *pull_up_down=None*, *initial=None*)

Add an input or output pin to the pin database.

An instance of *Pin* is created with the given arguments and added to the pin database *PinDB*.

Parameters

- **channel** (`int`) – GPIO channel number associated with the *Pin* to be added in the pin database.

- **gpio_function** (*int*) – Function of a GPIO channel: 1 (*GPIO.INPUT*) or 0 (*GPIO.OUTPUT*).
- **pull_up_down** (*int* or *None*, *optional*) – Initial value of an input channel, e.g. *GPIO.PUP_UP*. Default value is *None*.
- **initial** (*int* or *None*, *optional*) – Initial value of an output channel, e.g. *GPIO.HIGH*. Default value is *None*.

display_leds()

Simulate LEDs on an RPi by blinking small circles on a terminal.

In order to simulate LEDs turning on/off on an RPi, small circles are blinked on the terminal along with their GPIO pin number.

When a LED is turned on, it is shown as a small red circle on the terminal.

Example: terminal output

```
o [11]   o [9]   o [10]
```

where each circle represents a LED (here they are all turned off) and the number between brackets is the associated GPIO pin number.

Note: If `enable_printing` is set to *True*, the terminal's cursor will be hidden. It will be eventually shown again in `cleanup()` which is called by the main program when it is exiting.

The reason is to avoid messing with the display of LEDs by the displaying thread `th_display_leds`.

Important: `display_leds()` should be run by a thread and eventually stopped from the main thread by setting its `do_run` attribute to *False* to let the thread exit from its target function.

For example:

```
th = threading.Thread(target=self.display_leds, args=())
th.start()

# Your other code ...

# Time to stop thread
th.do_run = False
th.join()
```

static get_key_name(key)

Get the name of a keyboard key as a string.

The name of the special or alphanumeric key is given by the package `pynput`.

Parameters *key* (`pynput.keyboard.Key` or `pynput.keyboard.KeyCode`) – The keyboard key (from `pynput.keyboard`) whose name will be returned.

Returns *key_name* – Returns the name of the given keyboard key if one was found by `pynput`. Otherwise, it returns *None*.

Return type *str* or *None*

on_press(key)

When a valid keyboard key is pressed, set its state to *GPIO.LOW*.

Callback invoked from the thread `GPIO.Manager.th_listener`.

This thread is used to monitor the keyboard for any valid pressed key. Only keys defined in the pin database are treated, i.e. keys that were configured with `GPIO.setup()` are further processed.

Once a valid key is detected as pressed, its state is changed to `GPIO.LOW`.

Parameters `key` (`pynput.keyboard.Key`, `pynput.keyboard.KeyCode`, or `None`) –

The key parameter passed to callbacks is

- a `pynput.keyboard.Key` for special keys,
- a `pynput.keyboard.KeyCode` for normal alphanumeric keys, or
- `None` for unknown keys.

Ref.: <https://bit.ly/3k4whEs>

on_release (`key`)

When a valid keyboard key is released, set its state to `GPIO.HIGH`.

Callback invoked from the thread `GPIO.Manager.th_listener`.

This thread is used to monitor the keyboard for any valid released key. Only keys defined in the pin database are treated, i.e. keys that were configured with `GPIO.setup()` are further processed.

Once a valid key is detected as released, its state is changed to `GPIO.HIGH`.

Parameters `key` (`pynput.keyboard.Key`, `pynput.keyboard.KeyCode`, or `None`) –

The key parameter passed to callbacks is

- a `pynput.keyboard.Key` for special keys,
- a `pynput.keyboard.KeyCode` for normal alphanumeric keys, or
- `None` for unknown keys.

Ref.: <https://bit.ly/3k4whEs>

update_keymap (`new_keymap`)

Update the default dictionary mapping keys and GPIO channels.

`new_keymap` is a dictionary mapping some keys to their new GPIO channels, and will be used to update the default key-channel mapping defined in `SimulRPI.mapping`.

Parameters `new_keymap` (`dict`) – Dictionary that maps keys (`str`) to their new GPIO channels (`int`).

For example:

```
"key_to_channel_map":
{
    "f": 24,
    "g": 25,
    "h": 23
}
```

Note: If a key is associated to a channel that is already taken by another key, both keys' channels will be swapped. However, if a key is being linked to a `None` channel, then it will take on the maximum channel number available + 1.

static validate_key (`key`)

Validate if a key is recognized by `pynput`

A valid key can either be:

- a `pynput.keyboard.Key` for special keys (e.g. `tab` or `up`), or
- a `pynput.keyboard.KeyCode` for normal alphanumeric keys.

Parameters `key` (`str`) – The key (e.g. `'tab'`) that will be validated.

Returns `retval` – Returns `True` if it's a valid key. Otherwise, it returns `False`.

Return type `bool`

References

pynput reference: <https://pynput.readthedocs.io/en/latest/keyboard.html#reference>

See also:

`SimulRPI.mapping` for a list of special keys supported by pynput.

class `GPIO.Pin` (`channel`, `gpio_function`, `key=None`, `pull_up_down=None`, `initial=None`)

Bases: `object`

Class that represents a GPIO pin.

Parameters

- **channel** (`int`) – GPIO channel number based on the numbering system you have specified (`BOARD` or `BCM`).
- **gpio_function** (`int`) – Function of a GPIO channel: 1 (`GPIO.INPUT`) or 0 (`GPIO.OUTPUT`).
- **key** (`str` or `None`, *optional*) – Key associated with the GPIO channel, e.g. `"k"`.
- **pull_up_down** (`int` or `None`, *optional*) – Initial value of an input channel, e.g. `GPIO.PUP_UP`. Default value is `None`.
- **initial** (`int` or `None`, *optional*) – Initial value of an output channel, e.g. `GPIO.HIGH`. Default value is `None`.

Variables `state` (`int`) – State of the GPIO channel: 1 (`HIGH`) or 0 (`LOW`).

class `GPIO.PinDB`

Bases: `object`

Class for storing and modifying `Pins`.

Each instance of `GPIO.Pin` is saved in a dictionary that maps it to its channel number.

Note: The dictionary (a “database” of `Pins`) must be accessed through the different methods available in `PinDB`, e.g. `get_pin_from_channel()`.

create_pin (`channel`, `gpio_function`, `key=None`, `pull_up_down=None`, `initial=None`)

Create an instance of `GPIO.Pin` and save it in a dictionary.

Based on the given arguments, an instance of `GPIO.Pin` is created and added to a dictionary that acts like a database of pins with key being the pin's channel and the value is an instance of `Pin`.

Parameters

- **channel** (*int*) – GPIO channel number based on the numbering system you have specified (*BOARD* or *BCM*).
- **gpio_function** (*int*) – Function of a GPIO channel: 1 (*GPIO.INPUT*) or 0 (*GPIO.OUTPUT*).
- **key** (*str* or *None*, *optional*) – Key associated with the GPIO channel, e.g. “k”.
- **pull_up_down** (*int* or *None*, *optional*) – Initial value of an input channel, e.g. *GPIO.PUP_UP*. Default value is *None*.
- **initial** (*int* or *None*, *optional*) – Initial value of an output channel, e.g. *GPIO.HIGH*. Default value is *None*.

get_pin_from_channel (*channel*)

Get a *Pin* from a given channel.

Parameters **channel** (*int*) – GPIO channel number associated with the *Pin* to be retrieved.

Returns **Pin** – If no *Pin* could be retrieved based on the given channel, *None* is returned. Otherwise, a *Pin* object is returned.

Return type *GPIO.Pin* or *None*

get_pin_from_key (*key*)

Get a *Pin* from a given pressed/released key.

Parameters **key** (*str*) – The pressed/released key that is associated with the *Pin* to be retrieved.

Returns **Pin** – If no *Pin* could be retrieved based on the given key, *None* is returned. Otherwise, a *Pin* object is returned.

Return type *GPIO.Pin* or *None*

get_pin_state (*channel*)

Get a *Pin*’s state from a given channel.

The state associated with a *Pin* can either be 1 (*HIGH*) or 0 (*LOW*).

Parameters **channel** (*int*) – GPIO channel number associated with the *Pin* whose state is to be returned.

Returns **state** – If no *Pin* could be retrieved based on the given channel number, then *None* is returned. Otherwise, the *Pin*’s state is returned: 1 (*HIGH*) or 0 (*LOW*).

Return type *int* or *None*

set_pin_key_from_channel (*channel*, *key*)

Set a *Pin*’s key from a given channel.

A *Pin* is retrieved based on a given channel, then its *key* is set with *key*.

Parameters

- **channel** (*int*) – GPIO channel number associated with the *Pin* whose key will be set.
- **key** (*str*) – The new key that a *Pin* will be updated with.

Returns **retval** – Returns *True* if the *Pin* was successfully set with *key*. Otherwise, it returns *False*.

Return type *bool*

set_pin_state_from_channel (*channel*, *state*)

Set a *Pin*'s state from a given channel.

A *Pin* is retrieved based on a given channel, then its *state* is set with *state*.

Parameters

- **channel** (*int*) – GPIO channel number associated with the *Pin* whose state will be set.
- **state** (*int*) – State the GPIO channel should take: 1 (*HIGH*) or 0 (*LOW*).

Returns *retval* – Returns *True* if the *Pin* was successfully set with *state*. Otherwise, it returns *False* because the pin doesn't exist based on the given *channel*.

Return type *bool*

set_pin_state_from_key (*key*, *state*)

Set a *Pin*'s state from a given key.

A *Pin* is retrieved based on a given key, then its *state* is set with *state*.

Parameters

- **key** (*str*) – The key associated with the *Pin* whose state will be set.
- **state** (*int*) – State the GPIO channel should take: 1 (*HIGH*) or 0 (*LOW*).

Returns *retval* – Returns *True* if the *Pin* was successfully set with *state*. Otherwise, it returns *False* because the pin doesn't exist based on the given *key*.

Return type *bool*

GPIO.cleanup ()

Clean up any resources (e.g. GPIO channels).

At the end of any program, it is good practice to clean up any resources you might have used. This is no different with `RPI.GPIO`. By returning all channels you have used back to inputs with no pull up/down, you can avoid accidental damage to your RPi by shorting out the pins. [Ref: [RPI.GPIO wiki](#)]

Also, the two threads responsible for displaying “LEDs” on the terminal and listening for pressed/released keys are stopped.

Note: On an RPi, `cleanup()` will:

- only clean up GPIO channels that your script has used
- also clear the pin numbering system in use (*BOARD* or *BCM*)

Ref.: [RPI.GPIO wiki](#)

When using the package `SimulRPI`, `cleanup()` will:

- stop the displaying thread `Manager.th_display_leds`
 - stop the listener thread `Manager.th_listener`
 - show the cursor again which was hidden in `Manager.display_leds()`
 - reset the `GPIO.manager`'s attributes (an instance of `Manager`)
-

GPIO.input (*channel*)

Read the value of a GPIO pin.

Parameters **channel** (*int*) – Input GPIO channel number based on the numbering system you have specified (*BOARD* or *BCM*).

Returns state – If no *Pin* could be retrieved based on the given channel number, then `None` is returned. Otherwise, the *Pin*'s state is returned: 1 (*HIGH*) or 0 (*LOW*).

Return type `int` or `None`

Note: The listener thread (for monitoring pressed key) is started if it is not alive, i.e. it is not already running.

`GPIO.output(channel, state)`

Set the output state of a GPIO pin.

Parameters

- **channel** (`int`) – Output GPIO channel number based on the numbering system you have specified (*BOARD* or *BCM*).
- **state** (`int`) – State of the GPIO channel: 1 (*HIGH*) or 0 (*LOW*).

Note: The displaying thread (for showing “LEDs” on the terminal) is started if it is not alive, i.e. it is not already running.

`GPIO.setkeymap(key_to_channel_map)`

Set the keymap dictionary with new keys and channels.

The default dictionary `default_key_to_channel_map` (defined in *SimulRPI.mapping*) that maps keyboard keys to GPIO channels can be modified by providing your own mapping `key_to_channel_map` containing only the keys and channels that you want to be modified.

Parameters **key_to_channel_map** (`dict`) – A dictionary mapping keys (`str`) and GPIO channels (`int`) that will be used to update the default keymap found in *SimulRPI.mapping*.

For example:

```
key_to_channel_map:
{
    "q": 23,
    "w": 24,
    "e": 25
}
```

`GPIO.setmode(mode)`

Set the numbering system used to identify the I/O pins on an RPi within *RPi.GPIO*.

There are two ways of numbering the I/O pins on a Raspberry Pi within *RPi.GPIO*:

1. The *BOARD* numbering system: refers to the pin numbers on the P1 header of the Raspberry Pi board
2. The *BCM* numbers: refers to the channel numbers on the Broadcom SOC.

Parameters **mode** (`int`) – Numbering system used to identify the I/O pins on an RPi: *BOARD* or *BCM*.

References

Function description and more info from [RPI.GPIO wiki](#).

`RPI.GPIO.setprinting(enable_printing)`

Enable printing on the terminal.

If printing is enabled, small blinking red circles will be shown on the terminal, simulating LEDs connected to a Raspberry Pi. Otherwise, nothing will be printed on the terminal.

Parameters `enable_printing` (`bool`) – Whether to enable printing on the terminal.

`RPI.GPIO.setup(channel, gpio_function, pull_up_down=None, initial=None)`

Setup a GPIO channel as an input or output.

To configure a channel as an input:

```
RPI.GPIO.setup(channel, RPI.GPIO.IN)
```

To configure a channel as an output:

```
RPI.GPIO.setup(channel, RPI.GPIO.OUT)
```

You can also specify an initial value for your output channel:

```
RPI.GPIO.setup(channel, RPI.GPIO.OUT, initial=RPI.GPIO.HIGH)
```

Parameters

- **channel** (`int`) – GPIO channel number based on the numbering system you have specified (*BOARD* or *BCM*).
- **gpio_function** (`int`) – Function of a GPIO channel: 1 (*GPIO.INPUT*) or 0 (*GPIO.OUTPUT*).
- **pull_up_down** (`int` or `None`, *optional*) – Initial value of an input channel, e.g. *GPIO.PUP_UP*. Default value is `None`.
- **initial** (`int` or `None`, *optional*) – Initial value of an output channel, e.g. *GPIO.HIGH*. Default value is `None`.

References

[RPI.GPIO wiki](#)

`RPI.GPIO.setwarnings(show_warnings)`

Set warnings when configuring a GPIO pin other than the default (input).

It is possible that you have more than one script/circuit on the GPIO of your Raspberry Pi. As a result of this, if RPI.GPIO detects that a pin has been configured to something other than the default (input), you get a warning when you try to configure a script. [**Ref:** [RPI.GPIO wiki](#)]

Parameters `show_warnings` (`bool`) – Whether to show warnings when using a pin other than the default GPIO function (input).

2.2 SimulRPI.mapping

Module that defines the *dictionary* that maps keys and GPIO channels.

This module defines the default mapping between keyboard keys and GPIO channels. It is used by *GPIO* when monitoring the keyboard with the package *pynput* for any pressed/released key as to simulate a push button connected to a Raspberry Pi.

Notes

In early RPi models, there are 17 GPIO channels and in late RPi models, there are 28 GPIO channels.

By default, 28 GPIO channels (from 0 to 27) are mapped to alphanumeric and special keys. See the *content of the default keymap*.

Here is the full list of special keys you can use with info about some of them (taken from *pynput reference*):

- `alt`
- `alt_gr`
- `alt_l`
- `alt_r`
- `backspace`
- `caps_lock`
- `cmd`: A generic command button. On PC platforms, this corresponds to the Super key or Windows key, and on Mac it corresponds to the Command key.
- `cmd_l`: The left command button. On PC platforms, this corresponds to the Super key or Windows key, and on Mac it corresponds to the Command key.
- `cmd_r`: The right command button. On PC platforms, this corresponds to the Super key or Windows key, and on Mac it corresponds to the Command key.
- `ctrl`: A generic Ctrl key.
- `ctrl_l`
- `ctrl_r`
- `delete`
- `down`
- `end`
- `enter`
- `esc`
- `f1`: The function keys. F1 to F20 are defined.
- `home`
- `insert`: The Insert key. This may be undefined for some platforms.
- `left`
- `media_next`
- `media_play_pause`

- `media_previous`
- `media_volume_down`
- `media_volume_mute`
- `media_volume_up`
- `menu`: The Menu key. This may be undefined for some platforms.
- `num_lock`: The NumLock key. This may be undefined for some platforms.
- `page_down`
- `page_up`
- `pause`: The Pause/Break key. This may be undefined for some platforms.
- `print_screen`: The PrintScreen key. This may be undefined for some platforms.
- `right`
- `scroll_lock`
- `shift`
- `shift_l`
- `shift_r`
- `space`
- `tab`
- `up`

References

- **RPi Header**: <https://bit.ly/30ZM2Uj>
- **pynput**: <https://pynput.readthedocs.io/>

Important:

- `GPIO.setkeymap()` allows you to modify the default keymap.
 - The keys for the default keymap `default_key_to_channel_map` must be strings and their values (the channels) should be integers.
-

Content of the default keymap dictionary (*key*: keyboard key as `string`, *value*: GPIO channel as `int`):

```
default_key_to_channel_map = {
    "0": 0,  # sudo on mac
    "1": 1,  # sudo on mac
    "2": 2,  # sudo on mac
    "3": 3,  # sudo on mac
    "4": 4,  # sudo on mac
    "5": 5,  # sudo on mac
    "6": 6,  # sudo on mac
    "7": 7,  # sudo on mac
    "8": 8,  # sudo on mac
    "9": 9,  # sudo on mac
    "q": 10, # sudo on mac
```

(continues on next page)

(continued from previous page)

```
"alt": 11, # left alt on mac
"alt_l": 12, # not recognized on mac
"alt_r": 13,
"alt_gr": 14,
"cmd": 15, # left cmd on mac
"cmd_l": 16, # not recognized on mac
"cmd_r": 17,
"ctrl": 18, # left ctrl on mac
"ctrl_l": 19, # not recognized on mac
"ctrl_r": 20,
"media_play_pause": 21,
"media_volume_down": 22,
"media_volume_mute": 23,
"media_volume_up": 24,
"shift": 25, # left shift on mac
"shift_l": 26, # not recognized on mac
"shift_r": 27,
}
```

Important: There are some platform limitations on using some of the keyboard keys with `pynput`.

For instance, on macOS some keyboard keys may require that you run your script with `sudo`. All alphanumeric keys and some special keys (e.g. backspace and right) require `sudo`. In the content of [default_key_to_channel_map](#) shown previously, I commented those keyboard keys that need `sudo` on macOS. The others don't need `sudo` on macOS such as `cmd_r` and `shift`.

For more information about those platform limitations, see [pynput documentation](#).

Warning: If you want to be able to run your python script with `sudo` in order to use some keys that require it, you might need to edit `/etc/sudoers` to add your `PYTHONPATH` if your script makes use of your `PYTHONPATH` as setup in the `~/.bashrc` file. However, I don't recommend editing `/etc/sudoers` since you might break your `sudo` command (e.g. `sudo: /etc/sudoers is owned by uid 501, should be 0`).

Instead, use the keys that don't require `sudo` such as `cmd_r` and `shift` on macOS.

Note: On macOS, if the left keys `alt_l`, `ctrl_l`, `cmd_l`, and `shift_l` are not recognized, use their generic counterparts instead: `alt`, `ctrl`, `cmd`, and `shift`.

2.3 SimulRPI.run_examples

Script for executing code examples on a Raspberry Pi or computer (simulation).

This script allows you to run different code examples on your Raspberry Pi (RPI) or computer in which case it will make use of the library `SimulRPI` which partly fakes `RPI.GPIO`.

The code examples test different parts of the library `SimulRPI` in order to show what it is capable of simulating from an RPI:

- Turn on/off LEDs
- Detect pressed button and perform an action

2.3.1 Usage

Once the **SimulRPI** package is installed, you should have access to the `run_examples` script:

```
$ run_examples -h

run_examples [-h] [-v] -e EXAMPLE_NUMBER [-m {BOARD,BCM}] [-s]
              [-l [LED_CHANNEL [LED_CHANNEL ...]]]
              [-b BUTTON_CHANNEL] [-k KEY_NAME]
              [-t TOTAL_TIME_BLINKING] [--on TIME_LED_ON]
              [--off TIME_LED_OFF]
```

Run the script on the RPi:

```
$ run_examples
```

Run the code for example **#1** on your computer using **SimulRPI.GPIO** which simulates **RPI.GPIO**: and default values for the options `-l` (channel 10) and `--on` (1 second):

```
$ run_examples -s -e 1
```

`run_examples.ex1_turn_on_led(channel, time_led_on=3)`

Example 1: Turn ON a LED for some specified time.

A LED will be turned on for `time_led_on` seconds.

Parameters

- **channel** (*int*) – Output GPIO channel number based on the numbering system you have specified (*BOARD* or *BCM*).
- **time_led_on** (*float*, *optional*) – Time in seconds the LED will stay turned ON. The default value is 3 seconds.

`run_examples.ex2_turn_on_many_leds(channels, time_led_on=3)`

Example 2: Turn ON multiple LEDs for some specified time.

All LEDs will be turned on for `time_led_on` seconds.

Parameters

- **channels** (*list*) – List of output GPIO channel numbers based on the numbering system you have specified (*BOARD* or *BCM*).
- **time_led_on** (*float*, *optional*) – Time in seconds the LEDs will stay turned ON. The default value is 3 seconds.

`run_examples.ex3_detect_button(channel)`

Example 3: Detect if a button is pressed.

The function waits for the button to be pressed associated with the given `channel`. As soon as the button is pressed, a message is printed and the function exits.

Parameters **channel** (*int*) – Input GPIO channel number based on the numbering system you have specified (*BOARD* or *BCM*).

Note: If the simulation mode is enabled (`-s`), the specified keyboard key will be detected if pressed. The keyboard key can be specified through the command line options `-b` (button channel) or `-k` (the key name, e.g. 'ctrl'). See *script's usage*.

```
run_examples.ex4_blink_led(channel, total_time_blinking=4, time_led_on=0.5, time_led_off=0.5)
```

Example 4: Blink a LED for some specified time.

The led will blink for a total of `total_time_blinking` seconds. The LED will stay turned on for `time_led_on` seconds before turning off for `time_led_off` seconds, and so on until `total_time_blinking` seconds elapse.

Press `ctrl + c` to stop the blinking completely and exit from the function.

Parameters

- **channel** (`int`) – Output GPIO channel number based on the numbering system you have specified (*BOARD* or *BCM*).
- **total_time_blinking** (`float`, *optional*) – Total time in seconds the LED will be blinking. The default value is 4 seconds.
- **time_led_on** (`float`, *optional*) – Time in seconds the LED will stay turned ON at a time. The default value is 0.5 seconds.
- **time_led_off** (`float`, *optional*) – Time in seconds the LED will stay turned OFF at a time. The default value is 0.5 seconds.

```
run_examples.ex5_blink_led_if_button(led_channel, button_channel, total_time_blinking=4,  
                                     time_led_on=0.5, time_led_off=0.5)
```

Example 5: If a button is pressed, blink a LED for some specified time.

As soon as the button from the given `button_channel` is pressed, the LED will blink for a total of `total_time_blinking` seconds.

The LED will stay turned on for `time_led_on` seconds before turning off for `time_led_off` seconds, and so on until `total_time_blinking` seconds elapse.

Press `ctrl + c` to stop the blinking completely and exit from the function.

Parameters

- **led_channel** (`int`) – Output GPIO channel number based on the numbering system you have specified (*BOARD* or *BCM*).
- **button_channel** (`int`) – Input GPIO channel number based on the numbering system you have specified (*BOARD* or *BCM*).
- **total_time_blinking** (`float`, *optional*) – Total time in seconds the LED will be blinking. The default value is 4 seconds.
- **time_led_on** (`float`, *optional*) – Time in seconds the LED will stay turned ON at a time. The default value is 0.5 seconds.
- **time_led_off** (`float`, *optional*) – Time in seconds the LED will stay turned OFF at a time. The default value is 0.5 seconds.

Note: If the simulation mode is enabled (*-s*), the specified keyboard key will be detected if pressed. The keyboard key can be specified through the command line options *-b* (button channel) or *-k* (the key name, e.g. 'ctrl'). See *script's usage*.

```
run_examples.main()
```

Main entry-point to the script.

According to the user's choice of action, the script might run one of the specified code examples.

If the simulation flag (*-s*) is used, then the module `SimulRPI.GPIO` will be used which partly fakes `RPI.GPIO`.

Notes

Only one action at a time can be performed.

`run_examples.setup_argparser()`

Setup the argument parser for the command-line script.

The script allows you to run a code example on your RPi or on your computer. In the latter case, it will make use of the module `SimulRPi.GPIO` which partly fakes `RPi.GPIO`.

Returns `args` – Simple class used by default by `parse_args()` to create an object holding attributes and return it¹.

Return type `argparse.Namespace`

References

2.4 SimulRPi.utils

Collection of utility functions used for the SimulRPi library.

`utils.blink_led(channel, time_led_on, time_led_off)`

Blink one LED.

A LED on the given `channel` will be turned ON and OFF for `time_led_on` seconds and `time_led_off` seconds, respectively.

Parameters

- **channel** (`int`) – Channel number associated with a LED which will blink.
- **time_led_on** (`float`) – Time in seconds the LED will stay turned ON at a time.
- **time_led_off** (`float`) – Time in seconds the LED will stay turned OFF at a time.

`utils.turn_off_led(channel)`

Turn off a LED from a given channel.

Parameters **channel** (`int`) – Channel number associated with a LED which will be turned off.

`utils.turn_on_led(channel)`

Turn on a LED from a given channel.

Parameters **channel** (`int`) – Channel number associated with a LED which will be turned on.

¹ `argparse.Namespace`.

INDICES AND TABLES

- `genindex`
- `modindex`

PYTHON MODULE INDEX

g

`GPIO`, [13](#)

r

`run_examples`, [24](#)

s

`SimulRPI.mapping`, [22](#)

u

`utils`, [27](#)

INDEX

A

`add_pin()` (*GPIO.Manager* method), 14

B

`blink_led()` (*in module utils*), 27

C

`cleanup()` (*in module GPIO*), 19

`create_pin()` (*GPIO.PinDB* method), 17

D

`display_leds()` (*GPIO.Manager* method), 15

E

`ex1_turn_on_led()` (*in module run_examples*), 25

`ex2_turn_on_many_leds()` (*in module run_examples*), 25

`ex3_detect_button()` (*in module run_examples*), 25

`ex4_blink_led()` (*in module run_examples*), 25

`ex5_blink_led_if_button()` (*in module run_examples*), 26

G

`get_key_name()` (*GPIO.Manager* static method), 15

`get_pin_from_channel()` (*GPIO.PinDB* method), 18

`get_pin_from_key()` (*GPIO.PinDB* method), 18

`get_pin_state()` (*GPIO.PinDB* method), 18

`GPIO`
module, 13

I

`input()` (*in module GPIO*), 19

M

`main()` (*in module run_examples*), 26

`Manager` (*class in GPIO*), 13

`module`
 GPIO, 13
 run_examples, 24

SimulRpi.mapping, 22

utils, 27

O

`on_press()` (*GPIO.Manager* method), 15

`on_release()` (*GPIO.Manager* method), 16

`output()` (*in module GPIO*), 20

P

`Pin` (*class in GPIO*), 17

`PinDB` (*class in GPIO*), 17

R

`run_examples`
module, 24

S

`set_pin_key_from_channel()` (*GPIO.PinDB* method), 18

`set_pin_state_from_channel()` (*GPIO.PinDB* method), 18

`set_pin_state_from_key()` (*GPIO.PinDB* method), 19

`setkeymap()` (*in module GPIO*), 20

`setmode()` (*in module GPIO*), 20

`setprinting()` (*in module GPIO*), 21

`setup()` (*in module GPIO*), 21

`setup_argparser()` (*in module run_examples*), 27

`setwarnings()` (*in module GPIO*), 21

`SimulRpi.mapping`
module, 22

T

`turn_off_led()` (*in module utils*), 27

`turn_on_led()` (*in module utils*), 27

U

`update_keymap()` (*GPIO.Manager* method), 16

`utils`
module, 27

V

`validate_key()` (*GPIO.Manager* static method), 16