rcpy Documentation

Release 0.2a

Mauricio de Oliveira

May 21, 2017

CONTENTS:

CHAPTER

ONE

INTRODUCTION

This package supports the hardware on the Robotics Cape running on a Beaglebone Black or a Beaglebone Blue.

1.1 Installation

See http://github.com/mcdeoliveira/rcpy#installation for installation instructions.

CHAPTER

TWO

MODULE *RCPY*

This module is responsible to loading and initializing all hardware devices associated with the Robotics Cape or Beagelbone Blue. Just type:

```
import rcpy
```

to load the module. After loading the Robotics Cape is left at the state `rcpy.PAUSED`. It will also automatically cleanup after you exit your program. You can add additional function to be called by the cleanup routine using `rcpy.add_cleanup()`.

2.1 Constants

rcpy.**IDLE**

rcpy.**RUNNING**

rcpy.**PAUSED**

rcpy.**EXITING**

2.2 Low-level functions

rcpy.**get_state**()

Get the current state, one of `rcpy.IDLE`, `rcpy.RUNNING`, `rcpy.PAUSED`, `rcpy.EXITING`.

rcpy.**set_state**(*state*)

Set the current state, *state* is one of `rcpy.IDLE`, `rcpy.RUNNING`, `rcpy.PAUSED`, `rcpy.EXITING`.

rcpy.**exit**()

Set state to `rcpy.EXITING`.

rcpy.**add_cleanup**(*fun*, *pars*)

Parameters

- **fun** – function to call at cleanup
- **pars** – list of positional parameters to pass to function *fun*

Add function *fun* and parameters *pars* to the list of cleanup functions.

CHAPTER

THREE

MODULE *RCPY.GPIO*

This module provides an interface to the GPIO pins used by the Robotics Cape. There are low level functions which closely mirror the ones available in the C library and also Classes that provide a higher level interface.

For example:

```
import rcpy.gpio as gpio
pause_button = gpio.Input(gpio.PAUSE_BTN)
```

imports the module and create an *`rcpy.gpio.Input`* object corresponding to the *PAUSE* button on the Robotics Cape. The command:

```
if pause_button.low():
  print('Got <PAUSE>!')
```

waits forever until the *PAUSE* button on the Robotics Cape is pressed and:

```
try:
    if pause_button.low(timeout = 2000):
        print('Got <PAUSE>!')
except gpio.InputTimeout:
    print('Timed out!')
```

waits for at most 2000 ms, i.e. 2 s, before giving up.

This module also provides the class *`rcpy.gpio.InputEvent`* to handle input events. For example:

```
class MyInputEvent(gpio.InputEvent):

    def action(self, event):
        print('Got <PAUSE>!')
```

defines a class that can be used to print `Got <PAUSE>!` every time an input event happens. To connect this class with the particular event that the *PAUSE* button is pressed instantiate:

```
pause_event = MyInputEvent(pause_button, gpio.InputEvent.LOW)
```

which will cause the method *action* of the *MyInputEvent* class be called every time the state of the *pause_button* becomes *`rcpy.gpio.LOW`*. The event handler must be started by calling:

```
pause_event.start()
```

and it can be stop by:

```
pause_event.stop()
```

Alternatively one could have created an input event handler by passing a function to the argument *target* of `rcpy.gpio.InputEvent` as in:

```
def pause_action(input, event):
    if event == gpio.InputEvent.LOW:
        print('<PAUSE> went LOW')
    elif event == gpio.InputEvent.HIGH:
        print('<PAUSE> went HIGH')

pause_event = gpio.InputEvent(pause_button,
                              gpio.InputEvent.LOW | gpio.InputEvent.HIGH,
                              target = pause_action)
```

Note that the function *pause_action* will be called when *pause_button* becomes either `rcpy.gpio.HIGH` or `rcpy.gpio.LOW` because the event passed to the the constructor `InputEvent` is:

```
gpio.InputEvent.LOW | gpio.InputEvent.HIGH
```

which is joined by the logical or operator |. The function *pause_action* decides on the type of event by checking the variable *event*. This event handler should be started and stopped using `rcpy.gpio.InputEvent.start()` and `rcpy.gpio.InputEvent.stop()` as before.

Additional positional or keyword arguments can be passed as in:

```
def pause_action_with_parameter(input, event, parameter):
    print('Got <PAUSE> with {}!'.format(parameter))

pause_event = gpio.InputEvent(pause_button, gpio.InputEvent.LOW,
                              target = pause_action_with_parameter,
                              vargs = ('some parameter',))
```

See also `rcpy.button.Button` for a better interface for working with the Robotics Cape buttons.

3.1 Constants

`rcpy.gpio.`**`HIGH`**
Logic high level; equals *1*.

`rcpy.gpio.`**`LOW`**
Logic low level; equals *0*.

`rcpy.gpio.`**`POLL_TIMEOUT`**
Timeout in ms to be used when polling GPIO input (Default 100ms)

`rcpy.gpio.`**`DEBOUNCE_INTERVAL`**
Interval in ms to be used for debouncing (Default 0.5ms)

3.2 Classes

class `rcpy.gpio.`**`InputTimeout`**
Exception representing an input timeout event.

class `rcpy.gpio.`**`Input`**(*pin*)

Parameters **`pin`** (*`int`*) – GPIO pin *rcpy.gpio.Input* represents one of the GPIO input pins in the Robotics Cape or Beaglebone Blue.

is_high()

Returns True if pin is equal to *rcpy.gpio.HIGH* and False if pin is *rcpy.gpio.LOW*

is_low()

Returns True if pin is equal to *rcpy.gpio.LOW* and False if pin is *rcpy.gpio.HIGH*

high_or_low(*debounce = 0*, *timeout = None*)

Parameters

- **debounce** (*int*) – number of times to read input for debouncing (default 0)
- **timeout** (*int*) – timeout in milliseconds (default None)

Raises ***rcpy.gpio.InputTimeout*** – if more than *timeout* ms have elapsed without the input changing

Returns the new state as *rcpy.gpio.HIGH* or *rcpy.gpio.LOW*

Wait for pin to change state.

If *timeout* is not None wait at most *timeout* ms.

If *timeout* is negative wait forever. This call cannot be interrupted.

If *timeout* is None wait forever by repeatedly polling in *rcpy.gpio.POLL_TIMEOUT* ms. This call can only be interrupted by calling *rcpy.exit()*.

high(*debounce = 0*, *timeout = None*)

Parameters

- **debounce** (*int*) – number of times to read input for debouncing (default 0)
- **timeout** (*int*) – timeout in milliseconds (default None)

Raises ***rcpy.gpio.InputTimeout*** – if more than *timeout* ms have elapsed without the input changing

Returns True if the new state is *rcpy.gpio.HIGH* and False if the new state is *rcpy.gpio.LOW*

Wait for pin to change state.

If *timeout* is not None wait at most *timeout* ms.

If *timeout* is negative wait forever. This call cannot be interrupted.

If *timeout* is None wait forever by repeatedly polling in *rcpy.gpio.POLL_TIMEOUT* ms. This call can only be interrupted by calling *rcpy.exit()*.

low(*debounce = 0*, *timeout = None*)

Parameters

- **debounce** (*int*) – number of times to read input for debouncing (default 0)
- **timeout** (*int*) – timeout in milliseconds (default None)

Raises ***rcpy.gpio.InputTimeout*** – if more than *timeout* ms have elapsed without the input changing

Returns True if the new state is *rcpy.gpio.LOW* and False if the new state is *rcpy.gpio.HIGH*

Wait for pin to change state.

If *timeout* is not `None` wait at most *timeout* ms.

If *timeout* is negative wait forever. This call cannot be interrupted.

If *timeout* is `None` wait forever by repeatedly polling in *rcpy.gpio.POLL_TIMEOUT* ms. This call can only be interrupted by calling *rcpy.exit()*.

class `rcpy.gpio.`**`InputEvent`**(*input*, *event*, *debounce = 0*, *timeout = None*, *target = None*, *vargs = ()*, *kwargs = {}*)

Bases threading.Thread

rcpy.gpio.InputEvent is an event handler for GPIO input events.

Parameters

- **`input`** (*int*) – instance of *rcpy.gpio.Input*
- **`event`** (*int*) – either *rcpy.gpio.InputEvent.HIGH* or *rcpy.gpio.InputEvent.LOW*
- **`debounce`** (*int*) – number of times to read input for debouncing (default 0)
- **`timeout`** (*int*) – timeout in milliseconds (default None)
- **`target`** (*int*) – callback function to run in case input changes (default None)
- **`vargs`** (*int*) – positional arguments for function *target* (default ())
- **`kwargs`** (*int*) – keyword arguments for function *target* (default {})

`LOW`

Event representing change to a low logic level.

`HIGH`

Event representing change to a high logic level.

`action`(*event*, **vargs*, ***kwargs*)

Parameters

- **`event`** – either *rcpy.gpio.HIGH* or *rcpy.gpio.LOW*
- **`vargs`** – variable positional arguments
- **`kwargs`** – variable keyword arguments

Action to perform when event is detected.

`start`()

Start the input event handler thread.

`stop`()

Attempt to stop the input event handler thread. Once it has stopped it cannot be restarted.

3.3 Low-level functions

`rcpy.gpio.`**`set`**(*pin*, *value*)

Parameters

- **`pin`** (*int*) – GPIO pin
- **`value`** (*int*) – value to set the pin (*rcpy.gpio.HIGH* or *rcpy.gpio.LOW*)

Raises **`rcpy.gpio.error`** – if it cannot write to *pin*

Set GPIO *pin* to the new *value*.

`rcpy.gpio.`**`get`**(*pin*)

Parameters **`pin`** (*`int`*) – GPIO pin

Raises **`rcpy.gpio.error`** – if it cannot read from *pin*

Returns the current value of the GPIO *pin*

This is a non-blocking call.

`rcpy.gpio.`**`read`**(*pin*, *timeout = None*)

Parameters

- **`pin`** (*`int`*) – GPIO pin
- **`timeout`** (*`int`*) – timeout in milliseconds (default None)

Raises

- **`rcpy.gpio.error`** – if it cannot read from *pin*
- ***`rcpy.gpio.InputTimeout`*** – if more than *timeout* ms have elapsed without the input changing

Returns the new value of the GPIO *pin*

Wait for value of the GPIO *pin* to change. This is a blocking call.

CHAPTER

FOUR

MODULE *RCPY.BUTTON*

This module provides an interface to the *PAUSE* and *MODE* buttons in the Robotics Cape. The command:

```
import rcpy.button as button
```

imports the module. The *Module rcpy.button* provides objects corresponding to the *PAUSE* and *MODE* buttons on the Robotics Cape. Those are *rcpy.button.pause* and *rcpy.button.mode*. For example:

```
if button.mode.pressed():
    print('<MODE> pressed!')
```

waits forever until the *MODE* button on the Robotics Cape is pressed and:

```
if button.mode.released():
    print('<MODE> released!')
```

waits forever until the *MODE* button on the Robotics Cape is released. Note that nothing will print if you first have to press the button before releasing because *rcpy.button.Button.released()* returns `False` after the first input event, which in this case was *pressed*. As with *Module rcpy.gpio*, it is possible to use *rcpy.gpio.InputTimeout* as in:

```
import rcpy.gpio as gpio
try:
    if button.mode.pressed(timeout = 2000):
        print('<MODE> pressed!')
except gpio.InputTimeout:
    print('Timed out!')
```

which waits for at most 2000 ms, i.e. 2 s, before giving up.

This module also provides the class *rcpy.button.ButtonEvent* to handle input events. For example:

```
class MyButtonEvent(button.ButtonEvent):

    def action(self, event):
        print('Got <PAUSE>!')
```

defines a class that can be used to print `Got <PAUSE>!` every time the *PAUSE* button is pressed. To instantiate and start the event handler use:

```
pause_event = MyButtonEvent(button.pause, button.ButtonEvent.PRESSED)
pause_event.start()
```

The event handler can be stop by calling:

```
pause_event.stop()
```

Alternatively one could have created an input event handler by passing a function to the argument *target* of `rcpy.button.ButtonEvent` as in:

```
def pause_action(input, event):
    if event == button.ButtonEvent.PRESSED:
        print('<PAUSE> pressed!')
    elif event == button.ButtonEvent.RELEASED:
        print('<PAUSE> released!')

pause_event = button.ButtonEvent(button.pause,
                                 button.ButtonEvent.PRESSED | button.ButtonEvent.
↪RELEASED,
                                 target = pause_action)
```

This event handler should be started and stopped using `rcpy.button.ButtonEvent.start()` and `rcpy.button.ButtonEvent.stop()` as in *Module rcpy.gpio*. Additional positional or keyword arguments can be passed as in:

```
def pause_action_with_parameter(input, event, parameter):
    print('Got <PAUSE> with {}!'.format(parameter))

pause_event = button.ButtonEvent(button.pause, button.ButtonEvent.PRESSED,
                                 target = pause_action_with_parameter,
                                 vargs = ('some parameter',))
```

The main difference between *Module rcpy.button* and *Module rcpy.gpio* is that *Module rcpy.button* defines the constants `rcpy.button.PRESSED` and `rcpy.button.RELEASED`, the events `rcpy.button.ButtonEvent.PRESSED` and `rcpy.button.ButtonEvent.RELEASED`, and its classes handle debouncing by default.

4.1 Constants

rcpy.button.**PRESSED**

State of a pressed button; equal to `rcpy.gpio.LOW`.

rcpy.button.**RELEASED**

State of a released button; equal to `rcpy.gpio.HIGH`.

rcpy.button.**pause**

`rcpy.button.Button` representing the Robotics Cape *PAUSE* button.

rcpy.button.**mode**

`rcpy.button.Button` representing the Robotics Cape *MODE* button.

rcpy.button.**DEBOUNCE**

Number of times to test for deboucing (Default 3)

4.2 Classes

class rcpy.button.**Button**

Bases `rcpy.gpio.Input`

`rcpy.button.Button` represents buttons in the Robotics Cape or Beaglebone Blue.

is_pressed(*debounce = rcpy.button.DEBOUNCE*, *timeout = None*)

Returns True if button state is equal to rcpy.gpio.PRESSED and False if pin is rcpy.gpio.RELEASED

is_released(*debounce = rcpy.button.DEBOUNCE*, *timeout = None*)

Returns True if button state is equal to rcpy.gpio.RELEASED and False if pin is rcpy.gpio.PRESSED

pressed_or_released(*debounce = rcpy.button.DEBOUNCE*, *timeout = None*)

Parameters

- **debounce** (*int*) – number of times to read input for debouncing (default *rcpy.button.DEBOUNCE*)
- **timeout** (*int*) – timeout in milliseconds (default None)

Raises ***rcpy.gpio.InputTimeout*** – if more than *timeout* ms have elapsed without the button state changing

Returns the new state as *rcpy.button.PRESSED* or *rcpy.button.RELEASED*

Wait for button state to change.

If *timeout* is not None wait at most *timeout* ms.

If *timeout* is negative wait forever. This call cannot be interrupted.

If *timeout* is None wait forever by repeatedly polling in *rcpy.gpio.POLL_TIMEOUT* ms. This call can only be interrupted by calling *rcpy.exit()*.

pressed(*debounce = rcpy.button.DEBOUNCE*, *timeout = None*)

Parameters

- **debounce** (*int*) – number of times to read input for debouncing (default *rcpy.button.DEBOUNCE*)
- **timeout** (*int*) – timeout in milliseconds (default None)

Raises ***rcpy.gpio.InputTimeout*** – if more than *timeout* ms have elapsed without the button state changing.

Returns True if the new state is *rcpy.button.PRESSED* and False if the new state is *rcpy.button.RELEASED*

Wait for button state to change.

If *timeout* is not None wait at most *timeout* ms.

If *timeout* is negative wait forever. This call cannot be interrupted.

If *timeout* is None wait forever by repeatedly polling in *rcpy.gpio.POLL_TIMEOUT* ms. This call can only be interrupted by calling *rcpy.exit()*.

released(*debounce = rcpy.button.DEBOUNCE*, *timeout = None*)

Parameters

- **debounce** (*int*) – number of times to read input for debouncing (default *rcpy.button.DEBOUNCE*)
- **timeout** (*int*) – timeout in milliseconds (default None)

Raises ***rcpy.gpio.InputTimeout*** – if more than *timeout* ms have elapsed without the button state changing.

Returns `True` if the new state is *`rcpy.button.RELEASED`* and `False` if the new state is *`rcpy.button.PRESSED`*

Wait for button state to change.

If *timeout* is not `None` wait at most *timeout* ms.

If *timeout* is negative wait forever. This call cannot be interrupted.

If *timeout* is `None` wait forever by repeatedly polling in *`rcpy.gpio.POLL_TIMEOUT`* ms. This call can only be interrupted by calling *`rcpy.exit()`*.

class `rcpy.button.`**`ButtonEvent`**(*input*, *event*, *debounce = rcpy.button.DEBOUNCE*, *timeout = None*, *target = None*, *vargs = ()*, *kwargs = {}*)

Bases *`rcpy.gpio.InputEvent`*

Parameters

- **`input`** (*`int`*) – instance of *`rcpy.gpio.Input`*
- **`event`** (*`int`*) – either *`rcpy.button.ButtonEvent.PRESSED`* or *`rcpy.button.ButtonEvent.RELEASED`*
- **`debounce`** (*`int`*) – number of times to read input for debouncing (default *rcpy.button.DEBOUNCE*)
- **`timeout`** (*`int`*) – timeout in milliseconds (default *None*)
- **`target`** (*`int`*) – callback function to run in case input changes (default *None*)
- **`vargs`** (*`int`*) – positional arguments for function *target* (default *()*)
- **`kwargs`** (*`int`*) – keyword arguments for function *target* (default *{}*)

`rcpy.button.ButtonEvent` is an event handler for button events.

`PRESSED`

Event representing pressing a button; equal to *`rcpy.gpio.InputEvent.LOW`*.

`RELEASED`

Event representing releasing a button; equal to *`rcpy.gpio.InputEvent.HIGH`*.

`action`(*event*, **vargs*, ***kwargs*)

Parameters

- **`event`** – either *`rcpy.button.PRESSED`* or *`rcpy.button.RELEASED`*
- **`vargs`** – variable positional arguments
- **`kwargs`** – variable keyword arguments

Action to perform when event is detected.

`start`()

Start the input event handler thread.

`stop`()

Attempt to stop the input event handler thread. Once it has stopped it cannot be restarted.

CHAPTER

FIVE

MODULE *RCPY.LED*

This module provides an interface to the *RED* and *GREEN* buttons in the Robotics Cape. The command:

```
import rcpy.led as led
```

imports the module. The *Module rcpy.led* provides objects corresponding to the *RED* and *GREEN* buttons on the Robotics Cape, namely `rcpy.led.red` and `rcpy.led.green`. For example:

```
led.red.on()
```

turns the *RED* LED on and:

```
led.green.off()
```

turns the *GREEN* LED off. Likewise:

```
led.green.is_on()
```

returns `True` if the *GREEN* LED is on and:

```
led.red.is_off()
```

returns `True` if the *RED* LED is off.

This module also provides the class `rcpy.led.Blink` to handle LED blinking. It spawns a thread that will keep LEDs blinking with a given period. For example:

```
blink = Blink(led.red, .5)
blink.start()
```

starts blinking the *RED* LED every 0.5 seconds. One can also instantiate an `rcpy.led.Blink` object by calling `rcpy.led.LED.blink()` as in:

```
blink = led.red.Blink(.5)
blink.start()
```

which produces the same result. One can stop or resume blinking by calling `rcpy.led.Blink.toggle()` as in:

```
blink.toggle()
```

or call:

```
blink.stop()
```

to permanently stop the blinking thread.

5.1 Constants

`rcpy.led.`**`ON`**

State of an on LED; equal to *`rcpy.gpio.HIGH`*.

`rcpy.led.`**`OFF`**

State of an off led; equal to *`rcpy.gpio.LOW`*.

`rcpy.led.`**`red`**

`rcpy.led.LED` representing the Robotics Cape *RED* LED.

`rcpy.led.`**`green`**

`rcpy.led.LED` representing the Robotics Cape *GREEN* LED.

5.2 Classes

class `rcpy.led.`**`LED`**(*output*, *state = rcpy.led.OFF*)

Bases `rcpy.gpio.Output`

Parameters

- **`output`** – GPIO pin
- **`state`** – initial LED state

`rcpy.led.LED` represents LEDs in the Robotics Cape or Beaglebone Blue.

`is_on`()

Returns `True` if LED is on and `False` if LED is off

`is_off`()

Returns `True` if LED is off and `False` if LED is on

`on`()

Change LED state to `rcpy.LED.ON`.

`off`()

Change LED state to `rcpy.LED.OFF`.

`toggle`()

Toggle current LED state.

`blink`(*period*)

Parameters **`period`** (*`float`*) – period of blinking

Returns an instance of *`rcpy.led.Blink`*.

Blinks LED with a period of *period* seconds.

class `rcpy.led.`**`Blink`**(*led*, *period*)

Bases threading.Thread

`set_period`(*period*)

Parameters **`period`** (*`float`*) – period of blinking

Set blinking period.

toggle()
Toggle blinking on and off. Call toggle again to resume or stop blinking.

start()
Start the blinking thread.

stop()
Stop the blinking thread. Blinking cannot resume after calling *rcpy.led.Blink.stop()*.

CHAPTER

SIX

MODULE *RCPY.ENCODER*

CHAPTER

SEVEN

MODULE *RCPY.MPU9250*

CHAPTER

EIGHT

MODULE *RCPY.MOTOR*

CHAPTER NINE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX