# Python 3 Interface for the Robotics Cape on the Beaglebone Black and the Beaglebone Blue

***Release 0.5***

**Mauricio C. de Oliveira**

**Sep 03, 2019**

# 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

# MODULES

## 2.1 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()*.

### 2.1.1 Constants

rcpy.**IDLE**

rcpy.**RUNNING**

rcpy.**PAUSED**

rcpy.**EXITING**

### 2.1.2 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.**idle**()
Set state to *rcpy.IDLE*.

rcpy.**run**()
Set state to *rcpy.RUNNING*.

rcpy.**pause**()
Set state to *rcpy.PAUSED*.

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.

## 2.2 Module *rcpy.adc*

This module provides an interface to the Analog-to-digital converters (ADCs) in the Robotics Cape. The BeagleBone's 12-bit ADC is used on the robotics cape for reading the LiPo battery voltage, the DC jack input voltage, and 4 auxiliary signals that the user can connect to the 6-pin JST SH header labelled ADC. The pinout of this header is as follows:

1. Ground
2. VDD_ADC (1.8V)
3. AIN0
4. AIN1
5. AIN2
6. AIN3

The command:

```
import rcpy.adc as adc
```

imports the module. The DC Jack and battery voltages can be read using:

```
adc.dc_jack.get_voltage()
```

and:

```
adc.battery.get_voltage()
```

which return a floating point representation of the battery and DC jack voltage. All 7 ADC channels on the Sitara can be read with:

```
rc_adc_raw(channel)
```

which returns the raw integer output of the 12-bit ADC.:

```
rc_adc_volt(channel)
```

additionally converts this raw value to a floating point voltage before returning. *channel* must be an integer from 0 to 6.

This module was contributed by Brendan Simon.

### 2.2.1 Constants

**CHANNEL_COUNT = 7**

**CHANNEL_MIN = 0**

**CHANNEL__MAX = 6**

rcpy.adc.**adc0**
*rcpy.adc.ADC* representing the Beaglebone *AIN 0*.

rcpy.adc.**adc1**
*rcpy.adc.ADC* representing the Beaglebone *AIN 1*.

rcpy.adc.**adc2**
*rcpy.adc.ADC* representing the Beaglebone *AIN 2*.

rcpy.adc.**adc3**
*rcpy.adc.ADC* representing the Beaglebone *AIN 3*.

rcpy.adc.**adc4**
*rcpy.adc.ADC* representing the Beaglebone *AIN 4*.

rcpy.adc.**adc5**
*rcpy.adc.ADC* representing the Beaglebone *AIN 5*.

rcpy.adc.**adc6**
*rcpy.adc.ADC* representing the Beaglebone *AIN 6*.

rcpy.adc.**adc7**
*rcpy.adc.ADC* representing the Beaglebone *AIN 7*.

### 2.2.2 Classes

**class** rcpy.adc.**ADC**

**Parameters** **channel** (*int*) – ADC channel

*rcpy.adc.ADC* represents one of the ADC channels in the Beaglebone.

**get_raw**()

**Returns** the raw integer output of the 12-bit ADC

**get_voltage**()

**Returns** the corresponding floating point voltage

**class** rcpy.adc.**DC_Jack**
*rcpy.adc.DC_Jack* represents the voltage in the Robotics Cape or Beaglebone Blue jack.

**get_voltage**()

**Returns** the corresponding floating point voltage

**class** rcpy.adc.**Battery**
*rcpy.adc.Battery* represents the voltage of the battery connected to the Robotics Cape or Beaglebone Blue.

**get_voltage**()

**Returns** the corresponding floating point voltage

## 2.3 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*. One can import those objects directly using:

```
from rcpy.button import mode, pause
```

After importing these objects:

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

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

```
if 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 would correspond to the GPIO line associated to the button going *rcpy.button.PRESSED*.

*rcpy.button.Button.pressed()* and *rcpy.button.Button.released()* also raise the exception *rcpy.gpio.InputTimeout* if the argument *timeout* is used as in:

```
import rcpy.gpio as gpio
try:
    if 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.

The methods *rcpy.button.Button.is_pressed()* and *rcpy.button.Button.is_released()* are *non-blocking* versions of the above methods. For example:

```
import time
while True:
    if mode.is_pressed():
        print('<Mode> pressed!')
        break
    time.sleep(1)
```

checks the status of the button *MODE* every 1 s and breaks when *MODE* is pressed. As with *Module rcpy.gpio*, a much better way to handle such events is to use event handlers, in the case of buttons the class *rcpy.button.ButtonEvent*. 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(pause, button.ButtonEvent.PRESSED)
pause_event.start()
```

Note that the event *rcpy.button.ButtonEvent.PRESSED* was used so that *MyButtonEvent.action* is called only when the *PAUSE* button is pressed. The event handler can be stopped 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:

```
from rcpy.button import ButtonEvent

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

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

This event handler should be started and stopped using *rcpy.button.ButtonEvent.start()* and *rcpy.button.ButtonEvent.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 = ButtonEvent(pause, 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.

### 2.3.1 Constants

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.**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.**DEBOUNCE**
: Number of times to test for deboucing (Default 3)

### 2.3.2 Classes

**class** rcpy.button.**Button**

**Bases** *rcpy.gpio.Input*

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

**is_pressed**()

**Returns** `True` if button state is equal to *rcpy.button.PRESSED* and `False` if line is *rcpy.button.RELEASED*

**is_released**()

**Returns** `True` if button state is equal to *rcpy.button.RELEASED* and `False` if line is *rcpy.button.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 otherwise wait forever until the input changes.

**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 otherwise wait forever until the input changes.

**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 otherwise wait forever until the input changes.

**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.

## 2.4 Module *rcpy.clock*

This module provides a class *rcpy.clock.Clock* that can be used to run actions periodically. Actions must inherit from the class *rcpy.clock.Action* and implement the method *rcpy.clock.Action.run()*. Examples of objects that inherit from *rcpy.clock.Action* are *rcpy.led.LED* and *rcpy.servo.Servo*.

For example:

```
import rcpy.clock as clock
from rcpy.led import red
clk = clock.Clock(red)
clk.start()
```

will start a *thread* that will blink the *red* LED every second. The command:

```
clk.stop()
```

will stop the LED blinking. The subclass *rcpy.led.Blink* will in addition turn off the led after stopping.

### 2.4.1 Classes

**class** rcpy.clock.**Action**

*rcpy.clock.Action* represents and action.

**run**()

Run the action.

**class** rcpy.clock.**Actions**(**actions*)

**Bases** *rcpy.clock.Action*

*rcpy.clock.Actions* represents a sequence of actions.

**Parameters** **actions** – comma separated list of actions.

**class** rcpy.clock.**Clock**(*action*, *period*)

**Bases** threading.Thread

*rcpy.clock.Clock* executes actions periodically.

**Parameters**

- **action** (Action) – the action to be executed
- **period** (*int*) – the period in seconds (default 1 second)

**set_period**(*period*)

**Parameters** **period** (*float*) – period in seconds

Set action period.

**toggle**()

Toggle action on and off. Call toggle again to resume or stop action.

**start**()

Start the action thread.

**stop**()

Stop the action thread. Action cannot resume after calling `rcpy.led.Blink.stop()`. Use *rcpy.clock.Clock.toggle()* to temporarily interrupt an action.

## 2.5 Module *rcpy.encoder*

This module provides an interface to the four *encoder channels* in the Robotics Cape. The command:

```
import rcpy.encoder as encoder
```

imports the module. The *Module rcpy.encoder* provides objects corresponding to the each of the encoder channels on the Robotics Cape, namely *rcpy.encoder.encoder1*, *rcpy.encoder.encoder2*, *rcpy.encoder.encoder3*, and *rcpy.encoder.encoder4*. It may be convenient to import one or all of these objects as in:

```
from rcpy.encoder import encoder2
```

The current encoder count can be obtained using:

```
encoder2.get()
```

One can also *reset* the count to zero using:

```
encoder2.reset()
```

or to an arbitrary count using:

```
encoder2.set(1024)
```

after which:

```
encoder2.get()
```

will return 1024.

### 2.5.1 Constants

rcpy.encoder.**encoder1**
*rcpy.encoder.Encoder* representing the Robotics Cape *Encoder 1*.

rcpy.encoder.**encoder2**
*rcpy.encoder.Encoder* representing the Robotics Cape *Encoder 2*.

rcpy.encoder.**encoder3**
*rcpy.encoder.Encoder* representing the Robotics Cape *Encoder 3*.

rcpy.encoder.**encoder4**
*rcpy.encoder.Encoder* representing the Robotics Cape *Encoder 4*.

### 2.5.2 Classes

**class** rcpy.encoder.**Encoder**(*channel*, *count* = *None*)

**Parameters**

- **output** – encoder channel (1 through 4)
- **state** – initial encoder count (Default None)

*rcpy.encoder.Encoder* represents encoders in the Robotics Cape or Beaglebone Blue.

**get**()

**Returns** current encoder count

**set**(*count*)
Set current encoder count to *count*.

**reset**()
Set current encoder count to *0*.

### 2.5.3 Low-level functions

rcpy.encoder.**get**(*channel*)

**Parameters** **channel** (*int*) – encoder channel number

**Returns** current encoder count

This is a non-blocking call.

rcpy.encoder.**set**(*channel*, *count = 0*)

**Parameters**

- **channel** (*int*) – encoder channel number
- **count** (*int*) – desired encoder count

Set encoder *channel* count to *value*.

## 2.6 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 becomes *rcpy.gpio.LOW*, which happens when it is pressed. The following slightly modified version:

```
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, which one can detect by catching the exception *rcpy.gpio.InputTimeout*.

In the same vein, the method *rcpy.gpio.Input.high()* would wait for the button *PAUSE* to become *rcpy.gpio.HIGH*. For instance:

```
if pause_button.high():
    print('<PAUSE> got high!')
```

waits forever until the *PAUSE* button on the Robotics Cape is released. Note that nothing will print if you first have to press the button before releasing because *rcpy.gpio.Input.high()* returns `False` after the first input event, which in this case would correspond to the GPIO pin going *rcpy.gpio.LOW*.

The methods *rcpy.gpio.Input.is_low()* and *rcpy.gpio.Input.is_high()* are *non-blocking* versions of the above methods. For example:

```
import time
while True:
    if pause_button.is_low():
        print('Got <PAUSE>!')
        break
    time.sleep(1)
```

checks the status of the button *PAUSE* every 1 s and breaks when *PAUSE* is pressed. A much better way to handle such events is to use the class *rcpy.gpio.InputEvent*. 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 stopped by:

```
pause_event.stop()
```

The event handler automatically runs on a separate *thread*, which means that the methods *rcpy.gpio.InputEvent.start()* and *rcpy.gpio.InputEvent.stop()* are *non-blocking*, that is they return immediately.

The class *rcpy.gpio.InputEvent* defines two types of events: *rcpy.gpio.InputEvent.HIGH* and *rcpy.gpio.InputEvent.LOW*. Note that these are not the same as *rcpy.gpio.HIGH* and *rcpy.gpio.LOW*! It is often convenient to import the base class *rcpy.gpio.InputEvent*:

```
from rcpy.gpio import InputEvent
```

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 == InputEvent.LOW:
        print('<PAUSE> went LOW')
    elif event == InputEvent.HIGH:
        print('<PAUSE> went HIGH')

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

The function *pause_action* will be called when *pause_button* generates either event *rcpy.gpio.InputEvent.HIGH* or *rcpy.gpio.InputEvent.LOW* because the events passed to the the constructor *InputEvent* were combined using the *logical or* operator |, as in:

```
InputEvent.LOW | InputEvent.HIGH,
```

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 = InputEvent(pause_button, 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.

### 2.6.1 Constants

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

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

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

### 2.6.2 Classes

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

**class** rcpy.gpio.**GPIO**(*chip*, *line*)

**Parameters**

- **chip** (*int*) – GPIO chip
- **line** (*int*) – GPIO line

*rcpy.gpio.GPIO* represents one of the GPIO input or output lines in the Robotics Cape or Beaglebone Blue. Users are not supposed to directly use this class. Use *rcpy.gpio.Input* or *rcpy.gpio.Output* instead.

**request**(*type*)

**Parameters** **type** (*int*) – the type of line as defined in libgpiod

Request GPIO chip and line.

**release**(*type*)
Release GPIO chip and line.

**class** rcpy.gpio.**Output**(*chip*, *line*)

**Bases** *rcpy.gpio.GPIO*

**Parameters**

- **chip** (*int*) – GPIO chip
- **line** (*int*) – GPIO line

*rcpy.gpio.Output* represents one of the GPIO output lines in the Robotics Cape or Beaglebone Blue.

**set**(*state*)

**Parameters** **state** (*int*) – set the state of the line, *rcpy.gpio.HIGH* or *rcpy.gpio.LOW*

**class** rcpy.gpio.**Input**(*chip*, *line*)

**Bases** *rcpy.gpio.GPIO*

**Parameters**

- **chip** (*int*) – GPIO chip
- **line** (*int*) – GPIO line

*rcpy.gpio.Input* represents one of the GPIO input lines in the Robotics Cape or Beaglebone Blue.

**get**()

**Returns** the state of the line, *rcpy.gpio.HIGH* or *rcpy.gpio.LOW* (non-blocking)

**is_high**()

**Returns** `True` if line is equal to *rcpy.gpio.HIGH* and `False` if line is *rcpy.gpio.LOW* (non-blocking)

**is_low**()

**Returns** `True` if line is equal to *rcpy.gpio.LOW* and `False` if line is *rcpy.gpio.HIGH* (non-blocking)

**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 line to change state. This is a blocking call.

If *timeout* is not `None` wait at most *timeout* ms otherwise wait forever until the input changes.

**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 line to change state. This is a blocking call.

If *timeout* is not `None` wait at most *timeout* ms otherwise wait forever until the input changes.

**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 line to change state. This is a blocking call.

If *timeout* is not `None` wait at most *timeout* ms otherwise wait forever until the input changes.

**read**(*line*, *timeout = None*)

**Parameters** **timeout** (*int*) – timeout in milliseconds (default None)

**Raises**

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

**Returns** the new value of the GPIO *line*

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

**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.

## 2.7 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*. It may be convenient to import one or all of these objects as in:

```
from rcpy.led import red, green
```

For example:

```
red.on()
```

turns the *RED* LED on and:

```
green.off()
```

turns the *GREEN* LED off. Likewise:

```
green.is_on()
```

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

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

starts blinking the *RED* LED every 0.5 seconds. 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.

One can also instantiate an *rcpy.led.Blink* object by calling *rcpy.led.LED.blink()* as in:

```
blink = red.blink(.5)
```

which returns an instance of *rcpy.led.Blink*. *rcpy.led.LED.blink()* automatically calls `rcpy.led.Blink.start()`.

### 2.7.1 Constants

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

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

`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*.

### 2.7.2 Classes

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

**Bases** *rcpy.gpio.Output*

**Parameters**

- **`output`** – GPIO line
- **`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** *rcpy.clock.Clock*

*rcpy.led.Blink* toggles led on and off periodically.

**Parameters**

- **led** (*LED*) – the led to toggle
- **period** (*int*) – the period in seconds

## 2.8 Module *rcpy.motor*

This module provides an interface to the four *motor channels* in the Robotics Cape. Those control a high power PWM (Pulse Width Modulation) signal which is typically used to control *DC Motors*. The command:

```
import rcpy.motor as motor
```

imports the module. The *Module rcpy.motor* provides objects corresponding to the each of the motor channels on the Robotics Cape, namely *rcpy.motor.motor1*, *rcpy.motor.motor2*, *rcpy.motor.motor3*, and *rcpy.motor.motor4*. It may be convenient to import one or all of these objects as in

```
from rcpy.motor import motor2
```

This is a convenience object which is equivalent to:

```
motor2 = Motor(2)
```

The current average voltage applied to the motor can be set using:

```
duty = 1
motor2.set(duty)
```

where *duty* is a number varying from -1 to 1 which controls the percentage of the voltage available to the Robotics Cape that should be applied on the motor. A motor can be turned off by:

```
motor2.set(0)
```

or using one of the special methods *rcpy.motor.Motor.free_spin()* or *rcpy.motor.Motor.brake()*, which can be used to turn off the motor and set it in a *free-spin* or *braking* configuration. For example:

```
motor2.free_spin()
```

puts *motor2* in *free-spin* mode. In *free-spin mode* the motor behaves as if there were no voltage applied to its terminal, that is it is allowed to spin freely. In *brake mode* the terminals of the motor are *short-circuited* and the motor winding will exert an opposing force if the motor shaft is moved. *Brake mode* is essentially the same as setting the duty cycle to zero.

### 2.8.1 Constants

rcpy.motor.**motor1**
: *rcpy.motor.Motor* representing the Robotics Cape *Motor 1*.

rcpy.motor.**motor2**
: *rcpy.motor.Motor* representing the Robotics Cape *Motor 2*.

rcpy.motor.**motor3**
: *rcpy.motor.Motor* representing the Robotics Cape *Motor 3*.

rcpy.motor.**motor4**
: *rcpy.motor.Motor* representing the Robotics Cape *Motor 4*.

### 2.8.2 Classes

**class** rcpy.motor.**Motor**(*channel*, *duty* = *None*)

**Parameters**

- **output** – motor channel (1 through 4)
- **state** – initial motor duty cycle (Default None)

*rcpy.motor.Motor* represents motors in the Robotics Cape or Beaglebone Blue.

**set**(*duty*)
Set current motor duty cycle to *duty*. *duty* is a number between -1 and 1.

**free_spin**()
Stops the motor and puts it in *free-spin* mode.

**brake**()
Stops the motor and puts it in *brake* mode.

### 2.8.3 Low-level functions

rcpy.motor.**set**(*channel*, *duty*)

**Parameters**

- **channel** (*int*) – motor channel number
- **duty** (*int*) – desired motor duty cycle

Sets the motor channel *channel* duty cycle to *duty*.

This is a non-blocking call.

rcpy.motor.**set_free_spin**(*channel*)

**Parameters** **channel** (*int*) – motor channel number

Puts the motor channel *channel* in *free-spin mode*.

This is a non-blocking call.

rcpy.motor.**set_brake**(*channel*)

**Parameters** **channel** (*int*) – motor channel number

Puts the motor channel *channel* in *brake mode*.

This is a non-blocking call.

## 2.9 Module *rcpy.mpu9250*

This module provides an interface to the on-board MPU-9250 Nine-Axis (Gyro + Accelerometer + Compass) MEMS device. The command:

```
import rcpy.mpu9250 as mpu9250
```

imports the module.

**IMPORTANT:** Beware that due to the way the Robotics Cape is written objects of the class *rcpy.mpu9250.IMU* are singletons, that is they all refer to the same instance.

Setup can be done at creation, such as in

```
imu = mpu9250.IMU(enable_dmp = True, dmp_sample_rate = 4,
                  enable_magnetometer = True)
```

which starts and initializes the MPU-9250 to use its DMP (Dynamic Motion Processor) to provide periodic readings at a rate of 4 Hz and also its magnetometer.

The data can be read using:

```
imu.read()
```

which performs a blocking call and can be used to synchronize execution with the DMP. For example:

```
while True:
    data = imu.read()
    print('heading = {}'.format(data['head']))
```

will print the current heading at a rate of 4 Hz. More details about the configuration options and the format of the data can be obtained in the help for the functions *rcpy.mpu9250.initialize()* and *rcpy.mpu9250.read()*.

### 2.9.1 Constants

The following constants can be used to set the accelerometer full scale register:

rcpy.mpu9250.**ACCEL_FSR_2G**

rcpy.mpu9250.**ACCEL_FSR_4G**

rcpy.mpu9250.**ACCEL_FSR_8G**

rcpy.mpu9250.**ACCEL_FSR_16G**

The following constants can be used to set the gyroscope full scale register:

rcpy.mpu9250.**GYRO_FSR_250DPS**

rcpy.mpu9250.**GYRO_FSR_500DPS**

rcpy.mpu9250.**GYRO_FSR_1000DPS**

rcpy.mpu9250.**GYRO_FSR_2000DPS**

The following constants can be used to set the accelerometer low-pass filter:

rcpy.mpu9250.**ACCEL_DLPF_OFF**

rcpy.mpu9250.**ACCEL_DLPF_184**

rcpy.mpu9250.**ACCEL_DLPF_92**

rcpy.mpu9250.**ACCEL_DLPF_41**

rcpy.mpu9250.**ACCEL_DLPF_20**

rcpy.mpu9250.**ACCEL_DLPF_10**

rcpy.mpu9250.**ACCEL_DLPF_5**

The following constants can be used to set the gyroscope low-pass filter:

rcpy.mpu9250.**GYRO_DLPF_OFF**

rcpy.mpu9250.**GYRO_DLPF_184**

rcpy.mpu9250.**GYRO_DLPF_92**

rcpy.mpu9250.**GYRO_DLPF_41**

rcpy.mpu9250.**GYRO_DLPF_20**

rcpy.mpu9250.**GYRO_DLPF_10**

rcpy.mpu9250.**GYRO_DLPF_5**

The following constants can be used to set the imu orientation:

rcpy.mpu9250.**ORIENTATION_Z_UP**

rcpy.mpu9250.**ORIENTATION_Z_DOWN**

rcpy.mpu9250.**ORIENTATION_X_UP**

rcpy.mpu9250.**ORIENTATION_X_DOWN**

rcpy.mpu9250.**ORIENTATION_Y_UP**

rcpy.mpu9250.**ORIENTATION_Y_DOWN**

rcpy.mpu9250.**ORIENTATION_X_FORWARD**

rcpy.mpu9250.**ORIENTATION_X_BACK**

### 2.9.2 Classes

**class** rcpy.mpu9250.**IMU**(***kwargs*)

**Parameters** **kwargs** (*kwargs*) – keyword arguments

rcpy.mpu9250.imu represents the MPU-9250 in the Robotics Cape or Beaglebone Blue.

Any keyword accepted by *rcpy.mpu9250.initialize()* can be given.

**set**(***kwargs*)

**Parameters** **kwargs** (*kwargs*) – keyword arguments

Set current MPU-9250 parameters.

Any keyword accepted by *rcpy.mpu9250.initialize()* can be given.

**read**()

**Returns** dictionary with current MPU-9250 measurements

Dictionary is constructed as in *rcpy.mpu9250.read()*.

### 2.9.3 Low-level functions

rcpy.mpu9250.**initialize**(*accel_fsr*, *gyro_fsr*, *accel_dlpf*, *gyro_dlpf*, *enable_magnetometer*, *orientation*, *compass_time_constant*, *dmp_interrupt_priority*, *dmp_sample_rate*, *show_warnings*, *enable_dmp*, *enable_fusion*)

**Parameters**

- **accel_fsr** (*int*) – accelerometer full scale
- **gyro_fsr** (*int*) – gyroscope full scale
- **accel_dlpf** (*int*) – accelerometer low-pass filter

- **gyro_dlpf** (*int*) – gyroscope low-pass filter
- **enable_magnetometer** (*bool*) – `True` enables the magnetometer
- **orientation** (*int*) – imu orientation
- **compass_time_constant** (*float*) – compass time-constant
- **dmp_interrupt_priority** (*int*) – DMP interrupt priority
- **dmp_sample_rate** (*int*) – DMP sample rate
- **show_warnings** (*int*) – `True` shows warnings
- **enable_dmp** (*bool*) – `True` enables the DMP
- **enable_fusion** (*bool*) – `True` enables data fusion algorithm

Configure and initialize the MPU-9250.

All parameters are optional. Default values are obtained by calling the `rc_get_default_imu_config()` from the Robotics Cape library.

`rcpy.mpu9250.`**`power_off`**`()`
Powers off the MPU-9250

`rcpy.mpu9250.`**`read_accel_data`**`()`

**Returns** list with three-axis acceleration in m/s$^2$

This function forces the MPU-9250 registers to be read.

`rcpy.mpu9250.`**`read_gyro_data`**`()`

**Returns** list with three-axis angular velocities in deg/s

This function forces the MPU-9250 registers to be read.

`rcpy.mpu9250.`**`read_mag_data`**`()`

**Raises** **mup9250Error** – if magnetometer is disabled

**Returns** list with 3D magnetic field vector in $\mu$T

This function forces the MPU-9250 registers to be read.

`rcpy.mpu9250.`**`read_imu_temp`**`()`

**Returns** the imu temperature in deg C

This function forces the MPU-9250 registers to be read.

`rcpy.mpu9250.`**`read`**`()`

**Returns** dictionary with the imu data; the keys in the dictionary depend on the current configuration

If the magnetometer is *enabled* the dictionary contains the following keys:

- **accel**: 3-axis accelerations (m/s$^2$)
- **gyro**: 3-axis angular velocities (degree/s)
- **mag**: 3D magnetic field vector in ($\mu$T)
- **quat**: orientation quaternion
- **tb**: pitch/roll/yaw X/Y/Z angles (radians)
- **head**: heading from magnetometer (radians)

If the magnetometer is *not enabled* the keys **mag** and **head** are not present.

This function forces the MPU-9250 registers to be read only if the DMP is disabled. Otherwise it returns the latest DMP data. It is a blocking call.

## 2.10 Module *rcpy.servo*

This module provides an interface to the eight *servo* and *ESC* (Electronic Speed Control) *channels* in the Robotics Cape. The 3-pin servo connectors are not polarized so make sure the black/brown ground wire is the one closest to the Robotics Cape. The command:

```
import rcpy.servo as servo
```

imports the module. The *Module rcpy.servo* provides objects corresponding to the each of the servo and ESC channels on the Robotics Cape, namely *rcpy.servo.servo1* through *rcpy.servo.servo8*, and namely *rcpy.servo.esc1* through *rcpy.servo.esc8*. It may be convenient to import one or all of these objects as in

```
from rcpy.servo import servo7
```

This is a convenience object which is equivalent to:

```
motor7 = Servo(7)
```

The position of the servo can be set using:

```
duty = 1.5
servo7.set(duty)
```

where *duty* is a number varying from -1.5 to 1.5, which controls the angle of the servo. This command does not send a pulse to the servo. In fact, before you can drive servos using the Robotics Cape you need to enable power to flow to servos using:

```
servo.enable()
```

For safety the servo power rail is disabled by default. Do not enable the servo power rail when using brushless ESCs as they can be damaged. Typical brushless ESCs only use the ground and signal lines, so if you need to control servos and ESC simultaneously simply cut or disconnect the middle power wire from the ESC connector. Use the command:

```
servo.disable()
```

to turn off the servo power rail.

Back to servos, you can set the duty *and* send a pulse to the servo at the same time using the command:

```
duty = 1.5
servo7.pulse(duty)
```

Alternatively, you can initiate a *rcpy.clock.Clock* object using:

```
period = 0.1
clk = servo7.start(period)
```

which starts sending the current servor *duty* value to the servo every 0.1s. The clock can be stopped by:

```
clk.stop()
```

The above is equivalent to:

```
import rcpy.clock as clock
period = 0.1
clk = clock.Clock(servo7, period)
clk.start()
```

Similar commands are available for ESC. For example:

```
from rcpy.servo import esc3
duty = 1.0
esc3.pulse(duty)
```

sends a *full-throttle* pulse to the ESC in channel 3.

### 2.10.1 Constants

rcpy.servo.**servo1**
: *rcpy.servo.Servo* representing the Robotics Cape *Servo 1*.

rcpy.servo.**servo2**
: *rcpy.servo.Servo* representing the Robotics Cape *Servo 2*.

rcpy.servo.**servo3**
: *rcpy.servo.Servo* representing the Robotics Cape *Servo 3*.

rcpy.servo.**servo4**
: *rcpy.servo.Servo* representing the Robotics Cape *Servo 4*.

rcpy.servo.**servo5**
: *rcpy.servo.Servo* representing the Robotics Cape *Servo 5*.

rcpy.servo.**servo6**
: *rcpy.servo.Servo* representing the Robotics Cape *Servo 6*.

rcpy.servo.**servo7**
: *rcpy.servo.Servo* representing the Robotics Cape *Servo 7*.

rcpy.servo.**servo8**
: *rcpy.servo.Servo* representing the Robotics Cape *Servo 8*.

rcpy.servo.**esc1**
: *rcpy.servo.ESC* representing the Robotics Cape *ESC 1*.

rcpy.servo.**esc2**
: *rcpy.servo.ESC* representing the Robotics Cape *ESC 2*.

rcpy.servo.**esc3**
: *rcpy.servo.ESC* representing the Robotics Cape *ESC 3*.

rcpy.servo.**esc4**
: *rcpy.servo.ESC* representing the Robotics Cape *ESC 4*.

rcpy.servo.**esc5**
: *rcpy.servo.ESC* representing the Robotics Cape *ESC 5*.

rcpy.servo.**esc6**
: *rcpy.servo.ESC* representing the Robotics Cape *ESC 6*.

rcpy.servo.**esc7**
: *rcpy.servo.ESC* representing the Robotics Cape *ESC 7*.

rcpy.servo.**esc8**
*rcpy.servo.ESC* representing the Robotics Cape *ESC 8*.

### 2.10.2 Classes

**class** rcpy.servo.**Servo**(*channel*, *duty* = *None*)

**Parameters**

- **output** – servo channel (1 through 4)
- **state** – initial servo duty cycle (Default 0)

*rcpy.servo.Servo* represents servos in the Robotics Cape or Beaglebone Blue.

**set**(*duty*)
Set current servo duty cycle to *duty*. *duty* is a number between -1.5 and 1.5. This method does not pulse the servo. Use *rcpy.servo.Servo.pulse()* for setting *and* pulsing the servo at the same time. Otherwise use a *rcpy.clock.Clock* or the method *rcpy.servo.Servo.start()* to create one. The value of *duty* corresponds to the following pulse widths and servo angles:

| input | width | angle | direction |
|---|---|---|---|
| -1.5 | 600us | 90 deg | counter-clockwise |
| -1.0 | 900us | 60 deg | counter-clockwise |
| 0.0 | 1500us | 0 deg | neutral |
| +1.0 | 2100us | 60 deg | clockwise |
| +1.5 | 2400us | 90 deg | clockwise |

**pulse**(*duty*)
Set current servo duty cycle to *duty* and send a single pulse to the servo.

**run**()
Send a single pulse to the servo using the current value of *duty*.

**start**(*period*)

**Parameters** **period**(*float*) – period

**Returns** an instance of *rcpy.clock.Clock*.

Pulses servo periodically every *period* seconds.

**class** rcpy.servo.**ESC**(*channel*, *duty* = *None*)

**Parameters**

- **output** – ESC channel (1 through 4)
- **state** – initial ESC duty cycle (Default 0)

***rcpy.servo.ESC* represents ESCs in the Robotics Cape** or Beaglebone Blue.

**set**(*duty*)
Set current ESC duty cycle to *duty*, in which *duty* is a number between -0.1 and 1.0. This method does not pulse the ESC. Use *rcpy.servo.ESC.pulse()* for setting *and* pulsing the ESC at the same time. Otherwise use a *rcpy.clock.Clock* or the method *rcpy.servo.ESC.start()* to create one. The value of *duty* corresponds to the following pulse widths and servo angles:

| input | width | power | status |
|---|---|---|---|
| -0.1 | 900us | armed | idle |
| 0.0 | 1000us | 0% | off |
| +0.5 | 1500us | 50% | half-throttle |
| +1.0 | 2000us | 100% | full-throttle |

**pulse**(*duty*)
Set current ESC duty cycle to *duty* and send a single pulse to the ESC.

**run**()
Send a single pulses to the ESC using the current value of *duty*.

**start**(*period*)

**Parameters** **period** (*float*) – period

**Returns** an instance of *rcpy.clock.Clock*.

Pulses ESC periodically every *period* seconds.

### 2.10.3 Low-level functions

rcpy.servo.**enable**()
Enables the servo power rail. Be careful when connecting ESCs!

rcpy.servo.**disable**()
Disables the servo power rail.

rcpy.servo.**pulse**(*channel*, *duty*)

**Parameters**

- **channel** (*int*) – servo channel number
- **duty** (*int*) – desired servo duty cycle

Sends a "normalized" pulse to the servo channel *channel*, in which *duty* is a float varying from *-1.5* to *1.5*. See *rcpy.servo.Servo.set()* for details.

This is a non-blocking call.

rcpy.servo.**pulse_all**(*channel*, *duty*)

**Parameters** **duty** (*int*) – desired servo duty cycle

Sends a "normalized" pulse to all servo channels, in which *duty* is a float varying from *-1.5* to *1.5*. See *rcpy.servo.Servo.set()* for details.

This is a non-blocking call.

rcpy.servo.**esc_pulse**(*channel*, *duty*)

**Parameters**

- **channel** (*int*) – servo channel number
- **duty** (*int*) – desired ESC duty cycle

Sends a "normalized" pulse to the ESC channel *channel*, in which *duty* is a float varying from *-0.1* to *1.5*. See *rcpy.servo.ESC.set()* for details.

This is a non-blocking call.

rcpy.servo.**esc_pulse_all**(*channel*, *duty*)

**Parameters** **duty** (*int*) – desired ESC duty cycle

Sends a "normalized" pulse to all ESC channels, in which *duty* is a float varying from *-0.1* to *1.5*. See *rcpy.servo.ESC.set()* for details.

This is a non-blocking call.

rcpy.servo.**oneshot_pulse**(*channel*, *duty*)

**Parameters**

- **channel** (*int*) – ESC channel number
- **duty** (*int*) – desired ESC duty cycle

Sends a "normalized" *oneshot* pulse to the ESC channel *channel*, in which *duty* is a float varying from *-0.1* to *1.0*.

This is a non-blocking call.

rcpy.servo.**oneshot_pulse_all**(*channel*, *duty*)

**Parameters** **duty** (*int*) – desired ESC duty cycle

Sends a "normalized" *oneshot* pulse to all ESC channels, in which *duty* is a float varying from *-0.1* to *1.0*.

This is a non-blocking call.

rcpy.servo.**pulse_us**(*channel*, *us*)

**Parameters**

- **channel** (*int*) – servo channel number
- **us** (*int*) – desired servo duty cycle is $\mu$ s

Sends a pulse of duration *us* $\mu$ s to all servo channels. This is a non-blocking call.

This is a non-blocking call.

rcpy.servo.**pulse_us_all**(*channel*, *us*)

**Parameters** **us** (*int*) – desired servo duty cycle is $\mu$ s

Sends a pulse of duration *us* $\mu$ s to all servo channels. This is a non-blocking call.

CHAPTER

THREE

# INDICES AND TABLES

- genindex
- modindex
- search

# PYTHON MODULE INDEX

## r

# INDEX

## H



## I



## L



## M



## O



## P



## R

## S



## T