Gpiozero Documentation

Release 1.3.1

Ben Nuttall

Mar 08, 2017

Contents

A simple interface to everyday GPIO components used with Raspberry Pi.

Created by Ben Nuttall of the Raspberry Pi Foundation, Dave Jones, and other contributors.

CHAPTER 1

About

Component interfaces are provided to allow a frictionless way to get started with physical computing:

```
from gpiozero import LED
from time import sleep

led = LED(17)

while True:
    led.on()
    sleep(1)
    led.off()
    sleep(1)
```

With very little code, you can quickly get going connecting your components together:

```
from gpiozero import LED, Button
from signal import pause

led = LED(17)
button = Button(3)

button.when_pressed = led.on
button.when_released = led.off

pause()
```

The library includes interfaces to many simple everyday components, as well as some more complex things like sensors, analogue-to-digital converters, full colour LEDs, robotics kits and more.

CHAPTER 2

Install

First, update your repositories list:

```
sudo apt-get update
```

Then install the package of your choice. Both Python 3 and Python 2 are supported. Python 3 is recommended:

```
sudo apt-get install python3-gpiozero
```

or:

```
sudo apt-get install python-gpiozero
```

CHAPTER 3

Documentation

Comprehensive documentation is available at https://gpiozero.readthedocs.io/.

CHAPTER 4

Development

This project is being developed on GitHub. Join in:

- Provide suggestions, report bugs and ask questions as issues
- Provide examples we can use as recipes
- Contribute to the code

Alternatively, email suggestions and feedback to mailto:ben@raspberrypi.org

CHAPTER 5

Contributors

- Ben Nuttall (project maintainer)
- Dave Jones
- Martin O'Hanlon
- Andrew Scheller
- Schelto vanDoorn

CHAPTER 6

Table of Contents

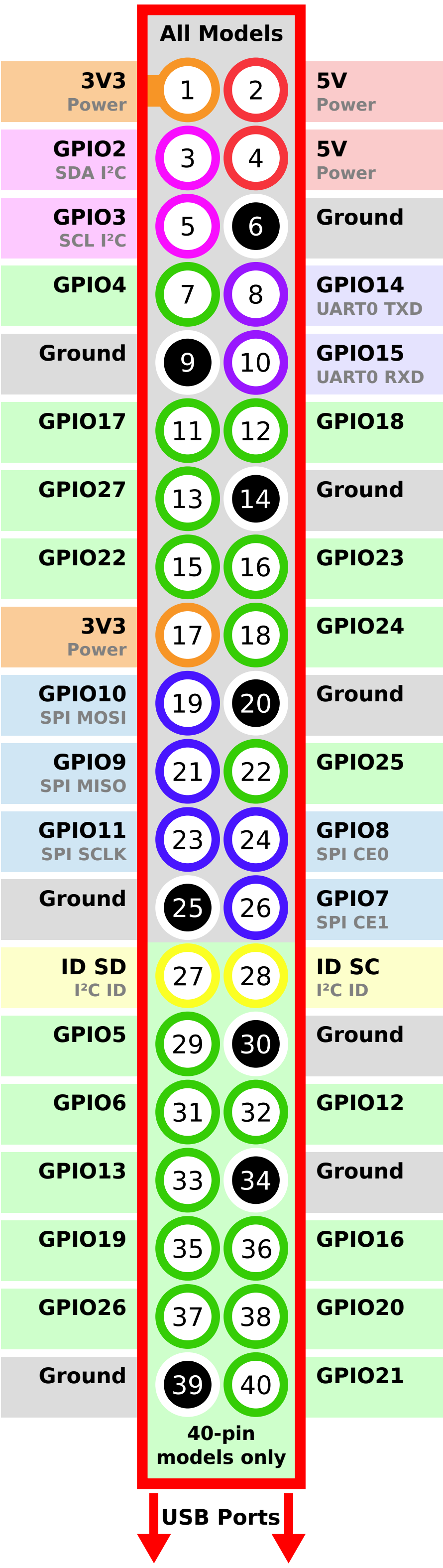
Recipes

The following recipes demonstrate some of the capabilities of the gpiozero library. Please note that all recipes are written assuming Python 3. Recipes *may* work under Python 2, but no guarantees!

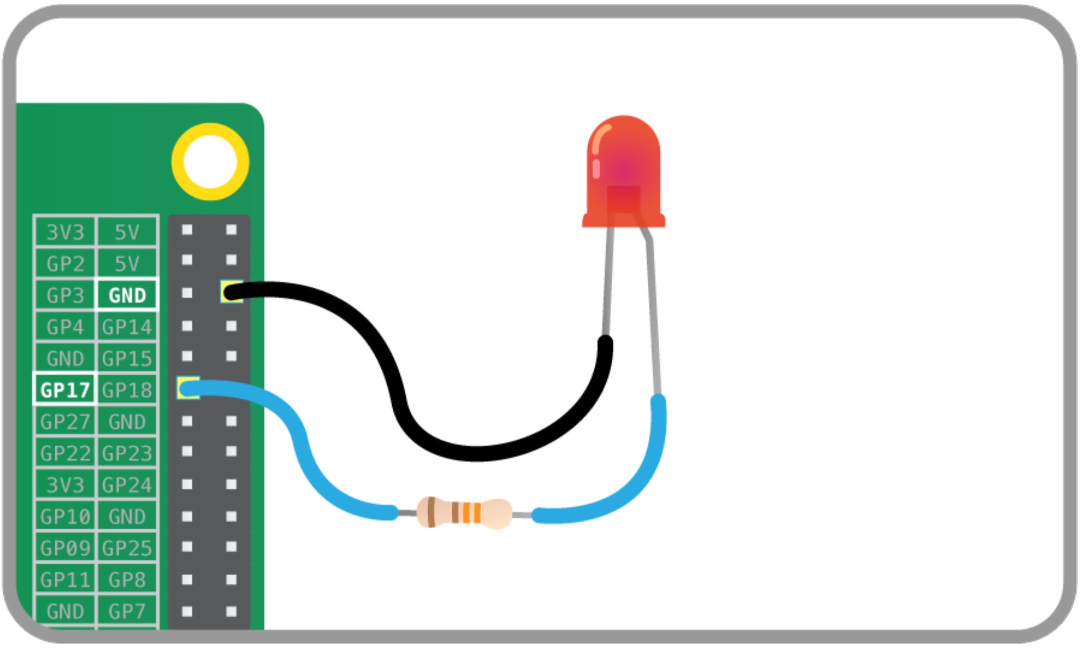
Pin Numbering

This library uses Broadcom (BCM) pin numbering for the GPIO pins, as opposed to physical (BOARD) numbering. Unlike in the RPi.GPIO library, this is not configurable.

Any pin marked "GPIO" in the diagram below can be used as a pin number. For example, if an LED was attached to "GPIO17" you would specify the pin number as 17 rather than 11:


All Models
3V3 Power
1
2
5V Power
GPIO2 SDA I²C
3
4
5V Power
GPIO3 SCL I²C
5
6
Ground
GPIO4
7
8
GPIO14 UART0 TXD
Ground
9
10
GPIO15 UART0 RXD
GPIO17
11
12
GPIO18
GPIO27
13
14
Ground
GPIO22
15
16
GPIO23
3V3 Power
17
18
GPIO24
GPIO10 SPI MOSI
19
20
Ground
GPIO9 SPI MISO
21
22
GPIO25
GPIO11 SPI SCLK
23
24
GPIO8 SPI CE0
Ground
25
26
GPIO7 SPI CE1
ID SD I²C ID
27
28
ID SC I²C ID
GPIO5
29
30
Ground
GPIO6
31
32
GPIO12
GPIO13
33
34
Ground
GPIO19
35
36
GPIO16
GPIO26
37
38
GPIO20
Ground
39
40
GPIO21
40-pin models only
USB Ports

LED



Turn an *LED* on and off repeatedly:

```
from gpiozero import LED
from time import sleep

red = LED(17)

while True:
    red.on()
    sleep(1)
    red.off()
    sleep(1)
```

Alternatively:

```
from gpiozero import LED
from signal import pause

red = LED(17)

red.blink()

pause()
```

Note: Reaching the end of a Python script will terminate the process and GPIOs may be reset. Keep your script alive with `signal.pause()`. See *Keep your script running* for more information.

LED with variable brightness

Any regular LED can have its brightness value set using PWM (pulse-width-modulation). In GPIO Zero, this can be achieved using *PWMLED* using values between 0 and 1:

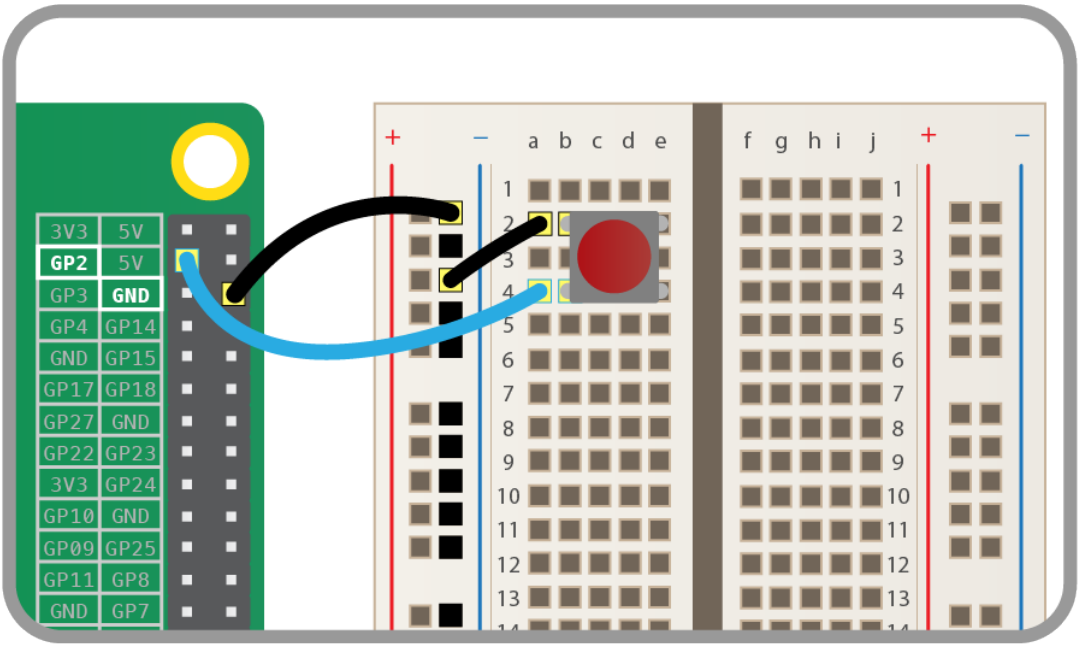
```
from gpiozero import PWMLED
from time import sleep
```

```
led = PWMLED(17)

while True:
    led.value = 0  # off
    sleep(1)
    led.value = 0.5  # half brightness
    sleep(1)
    led.value = 1  # full brightness
    sleep(1)
```

Similarly to blinking on and off continuously, a PWMLED can pulse (fade in and out continuously):

```
from gpiozero import PWMLED
from signal import pause

led = PWMLED(17)

led.pulse()

pause()
```

Button



Check if a *Button* is pressed:

```
from gpiozero import Button

button = Button(2)

while True:
    if button.is_pressed:
        print("Button is pressed")
    else:
        print("Button is not pressed")
```

Wait for a button to be pressed before continuing:

```python
from gpiozero import Button

button = Button(2)

button.wait_for_press()
print("Button was pressed")
```

Run a function every time the button is pressed:

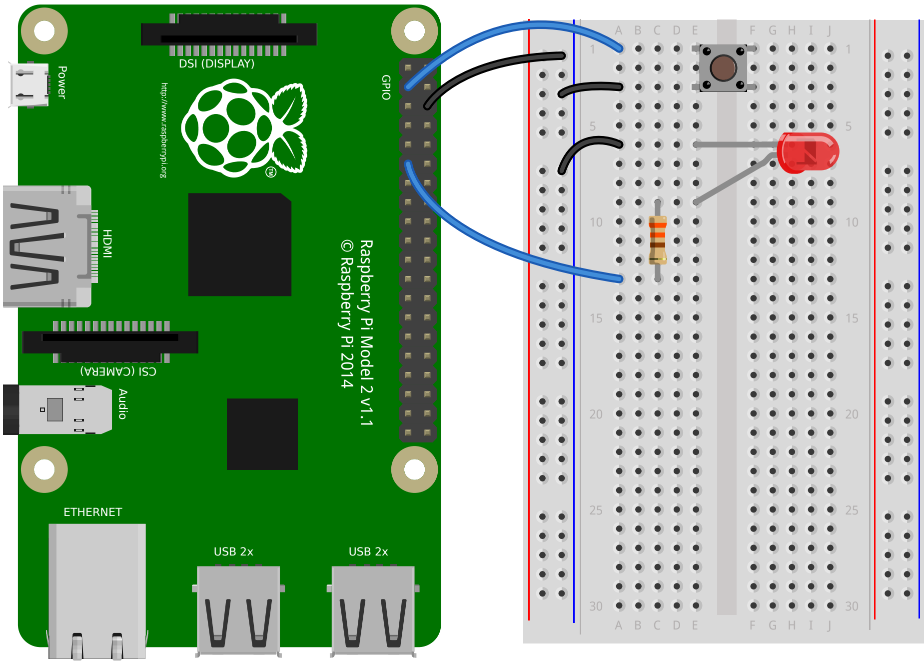
```python
from gpiozero import Button
from signal import pause

def say_hello():
    print("Hello!")

button = Button(2)

button.when_pressed = say_hello

pause()
```

Note: Note that the line `button.when_pressed = say_hello` does not run the function `say_hello`, rather it creates a reference to the function to be called when the button is pressed. Accidental use of `button.when_pressed = say_hello()` would set the `when_pressed` action to `None` (the return value of this function) which would mean nothing happens when the button is pressed.

Similarly, functions can be attached to button releases:

```python
from gpiozero import Button
from signal import pause

def say_hello():
    print("Hello!")

def say_goodbye():
    print("Goodbye!")

button = Button(2)

button.when_pressed = say_hello
button.when_released = say_goodbye

pause()
```

Button controlled LED



Turn on an *LED* when a *Button* is pressed:

```
from gpiozero import LED, Button
from signal import pause

led = LED(17)
button = Button(2)

button.when_pressed = led.on
button.when_released = led.off

pause()
```

Alternatively:

```
from gpiozero import LED, Button
from signal import pause

led = LED(17)
button = Button(2)

led.source = button.values

pause()
```

Button controlled camera

Using the button press to trigger `PiCamera` to take a picture using `button.when_pressed = camera.capture` would not work because the `capture()` method requires an `output` parameter. However, this can be achieved using a custom function which requires no parameters:

```
from gpiozero import Button
from picamera import PiCamera
from datetime import datetime
from signal import pause
```

```
button = Button(2)
camera = PiCamera()

def capture():
    datetime = datetime.now().isoformat()
    camera.capture('/home/pi/%s.jpg' % datetime)

button.when_pressed = capture

pause()
```

Another example could use one button to start and stop the camera preview, and another to capture:

```
from gpiozero import Button
from picamera import PiCamera
from datetime import datetime
from signal import pause

left_button = Button(2)
right_button = Button(3)
camera = PiCamera()

def capture():
    datetime = datetime.now().isoformat()
    camera.capture('/home/pi/%s.jpg' % datetime)

left_button.when_pressed = camera.start_preview
left_button.when_released = camera.stop_preview
right_button.when_pressed = capture

pause()
```

Shutdown button

The `Button` class also provides the ability to run a function when the button has been held for a given length of time. This example will shut down the Raspberry Pi when the button is held for 2 seconds:

```
from gpiozero import Button
from subprocess import check_call
from signal import pause

def shutdown():
    check_call(['sudo', 'poweroff'])

shutdown_btn = Button(17, hold_time=2)
shutdown_btn.when_held = shutdown

pause()
```

LEDBoard

A collection of LEDs can be accessed using `LEDBoard`:

```
from gpiozero import LEDBoard
from time import sleep
from signal import pause

leds = LEDBoard(5, 6, 13, 19, 26)
```

```
leds.on()
sleep(1)
leds.off()
sleep(1)
leds.value = (1, 0, 1, 0, 1)
sleep(1)
leds.blink()

pause()
```

Using *LEDBoard* with `pwm=True` allows each LED's brightness to be controlled:

```
from gpiozero import LEDBoard

leds = LEDBoard(5, 6, 13, 19, 26, pwm=True)

leds.value = (0.2, 0.4, 0.6, 0.8, 1.0)
```

LEDBarGraph

A collection of LEDs can be treated like a bar graph using *LEDBarGraph*:

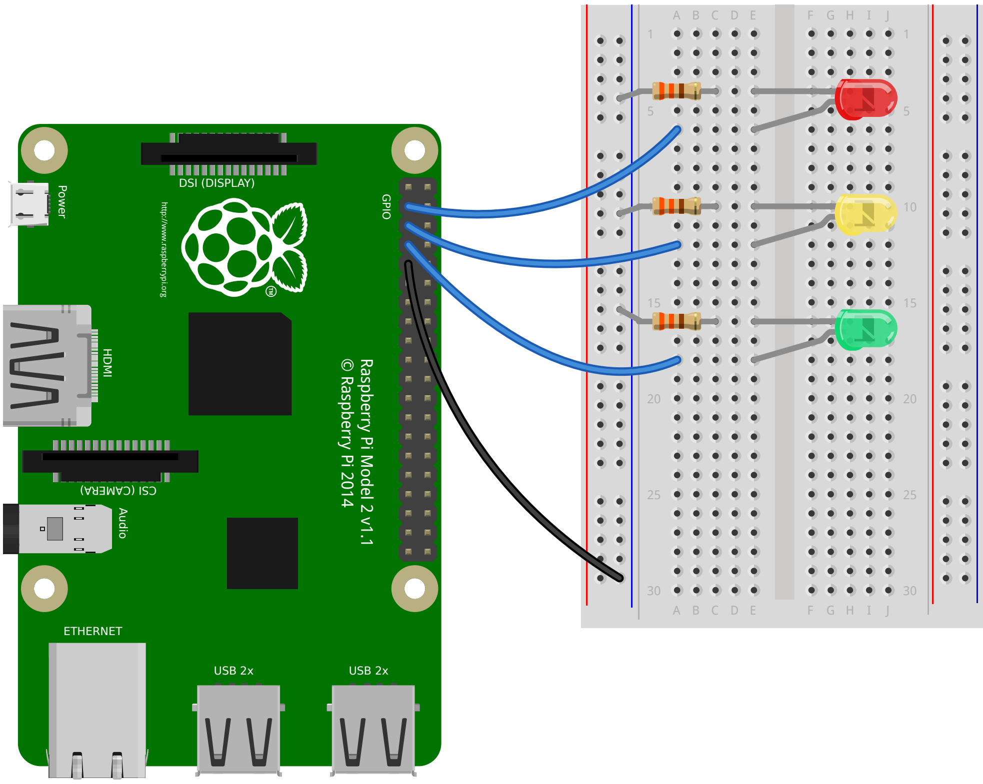
```
from gpiozero import LEDBarGraph
from time import sleep

graph = LEDBarGraph(5, 6, 13, 19, 26, pwm=True)

graph.value = 1/10  # (0.5, 0, 0, 0, 0)
sleep(1)
graph.value = 3/10  # (1, 0.5, 0, 0, 0)
sleep(1)
graph.value = -3/10  # (0, 0, 0, 0.5, 1)
sleep(1)
graph.value = 9/10  # (1, 1, 1, 1, 0.5)
sleep(1)
graph.value = 95/100  # (1, 1, 1, 1, 0.75)
sleep(1)
```

Note values are essentially rounded to account for the fact LEDs can only be on or off when `pwm=False` (the default).

However, using *LEDBarGraph* with `pwm=True` allows more precise values using LED brightness:

```
from gpiozero import LEDBarGraph
from time import sleep

graph = LEDBarGraph(5, 6, 13, 19, 26, pwm=True)

graph.value = 1/10  # (0.5, 0, 0, 0, 0)
sleep(1)
graph.value = 3/10  # (1, 0.5, 0, 0, 0)
sleep(1)
graph.value = -3/10  # (0, 0, 0, 0.5, 1)
sleep(1)
graph.value = 9/10  # (1, 1, 1, 1, 0.5)
sleep(1)
graph.value = 95/100  # (1, 1, 1, 1, 0.75)
sleep(1)
```

Traffic Lights



A full traffic lights system.

Using a *TrafficLights* kit like Pi-Stop:

```
from gpiozero import TrafficLights
from time import sleep

lights = TrafficLights(2, 3, 4)

lights.green.on()

while True:
    sleep(10)
    lights.green.off()
    lights.amber.on()
    sleep(1)
    lights.amber.off()
    lights.red.on()
    sleep(10)
    lights.amber.on()
    sleep(1)
    lights.green.on()
    lights.amber.off()
    lights.red.off()
```

Alternatively:

```
from gpiozero import TrafficLights
from time import sleep
from signal import pause

lights = TrafficLights(2, 3, 4)

def traffic_light_sequence():
```

```
    while True:
        yield (0, 0, 1) # green
        sleep(10)
        yield (0, 1, 0) # amber
        sleep(1)
        yield (1, 0, 0) # red
        sleep(10)
        yield (1, 1, 0) # red+amber
        sleep(1)

lights.source = traffic_light_sequence()

pause()
```

Using *LED* components:

```
from gpiozero import LED
from time import sleep

red = LED(2)
amber = LED(3)
green = LED(4)

green.on()
amber.off()
red.off()

while True:
    sleep(10)
    green.off()
    amber.on()
    sleep(1)
    amber.off()
    red.on()
    sleep(10)
    amber.on()
    sleep(1)
    green.on()
    amber.off()
    red.off()
```

Travis build LED indicator

Use LEDs to indicate the status of a Travis build. A green light means the tests are passing, a red light means the build is broken:

```
from travispy import TravisPy
from gpiozero import LED
from gpiozero.tools import negated
from time import sleep
from signal import pause

def build_passed(repo='RPi-Distro/python-gpiozero', delay=3600):
    t = TravisPy()
    r = t.repo(repo)
    while True:
        yield r.last_build_state == 'passed'
        sleep(delay) # Sleep an hour before hitting travis again

red = LED(12)
green = LED(16)
```

```
red.source = negated(green.values)
green.source = build_passed()
pause()
```

Note this recipe requires travispy. Install with `sudo pip3 install travispy`.

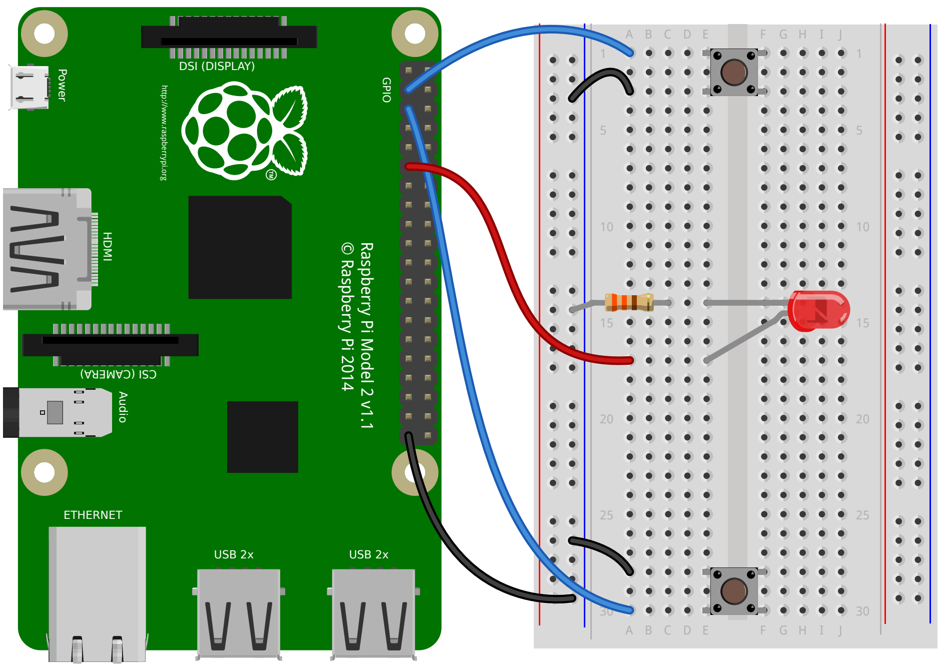
Push button stop motion

Capture a picture with the camera module every time a button is pressed:

```
from gpiozero import Button
from picamera import PiCamera

button = Button(2)
camera = PiCamera()

camera.start_preview()
frame = 1
while True:
    button.wait_for_press()
    camera.capture('/home/pi/frame%03d.jpg' % frame)
    frame += 1
```

See Push Button Stop Motion for a full resource.

Reaction Game



When you see the light come on, the first person to press their button wins!

```
from gpiozero import Button, LED
from time import sleep
import random

led = LED(17)

```

```
player_1 = Button(2)
player_2 = Button(3)

time = random.uniform(5, 10)
sleep(time)
led.on()

while True:
    if player_1.is_pressed:
        print("Player 1 wins!")
        break
    if player_2.is_pressed:
        print("Player 2 wins!")
        break

led.off()
```

See Quick Reaction Game for a full resource.

GPIO Music Box

Each button plays a different sound!

```
from gpiozero import Button
import pygame.mixer
from pygame.mixer import Sound
from signal import pause

pygame.mixer.init()

sound_pins = {
    2: Sound("samples/drum_tom_mid_hard.wav"),
    3: Sound("samples/drum_cymbal_open.wav"),
}

buttons = [Button(pin) for pin in sound_pins]
for button in buttons:
    sound = sound_pins[button.pin.number]
    button.when_pressed = sound.play

pause()
```

See GPIO Music Box for a full resource.

All on when pressed

While the button is pressed down, the buzzer and all the lights come on.

FishDish:

```
from gpiozero import FishDish
from signal import pause

fish = FishDish()

fish.button.when_pressed = fish.on
fish.button.when_released = fish.off

pause()
```

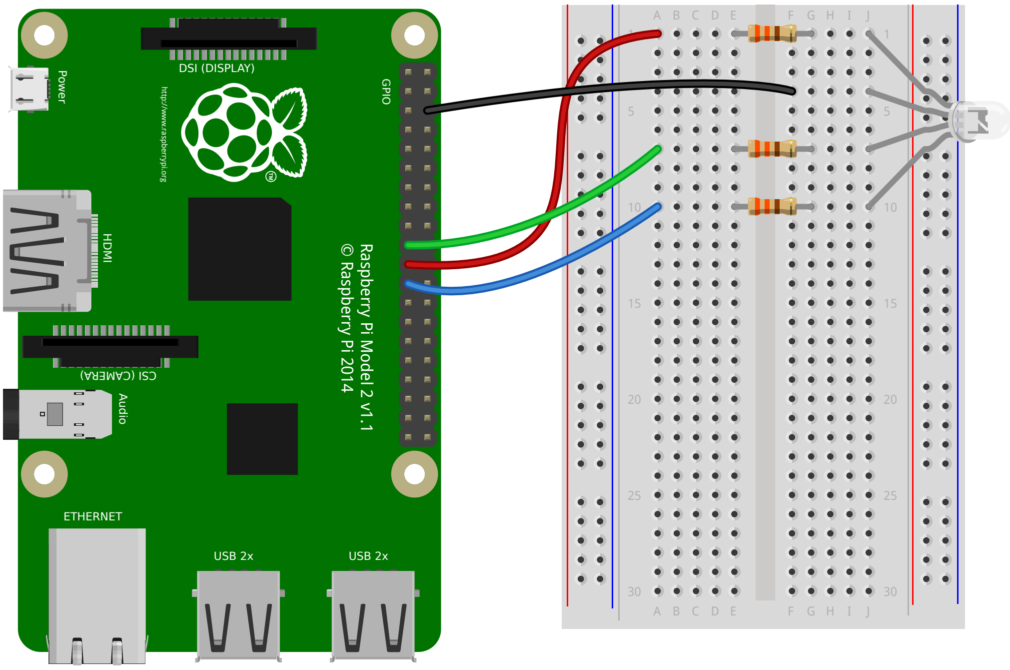
Ryanteck *TrafficHat*:

```
from gpiozero import TrafficHat
from signal import pause

th = TrafficHat()

th.button.when_pressed = th.on
th.button.when_released = th.off

pause()
```

Using *LED*, *Buzzer*, and *Button* components:

```
from gpiozero import LED, Buzzer, Button
from signal import pause

button = Button(2)
buzzer = Buzzer(3)
red = LED(4)
amber = LED(5)
green = LED(6)

things = [red, amber, green, buzzer]

def things_on():
    for thing in things:
        thing.on()

def things_off():
    for thing in things:
        thing.off()

button.when_pressed = things_on
button.when_released = things_off

pause()
```

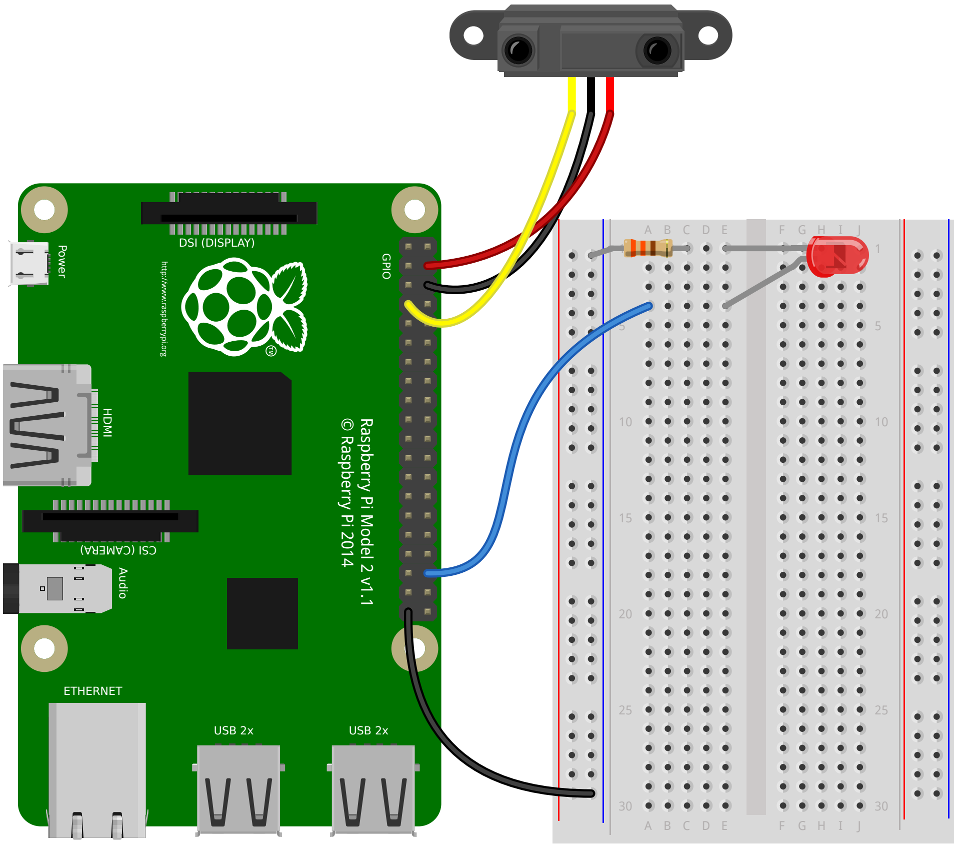
Full color LED



Making colours with an *RGBLED*:

```
from gpiozero import RGBLED
from time import sleep

led = RGBLED(red=9, green=10, blue=11)

led.red = 1  # full red
sleep(1)
led.red = 0.5  # half red
sleep(1)

led.color = (0, 1, 0)  # full green
sleep(1)
led.color = (1, 0, 1)  # magenta
sleep(1)
led.color = (1, 1, 0)  # yellow
sleep(1)
led.color = (0, 1, 1)  # cyan
sleep(1)
led.color = (1, 1, 1)  # white
sleep(1)

led.color = (0, 0, 0)  # off
sleep(1)

# slowly increase intensity of blue
for n in range(100):
    led.blue = n/100
    sleep(0.1)
```

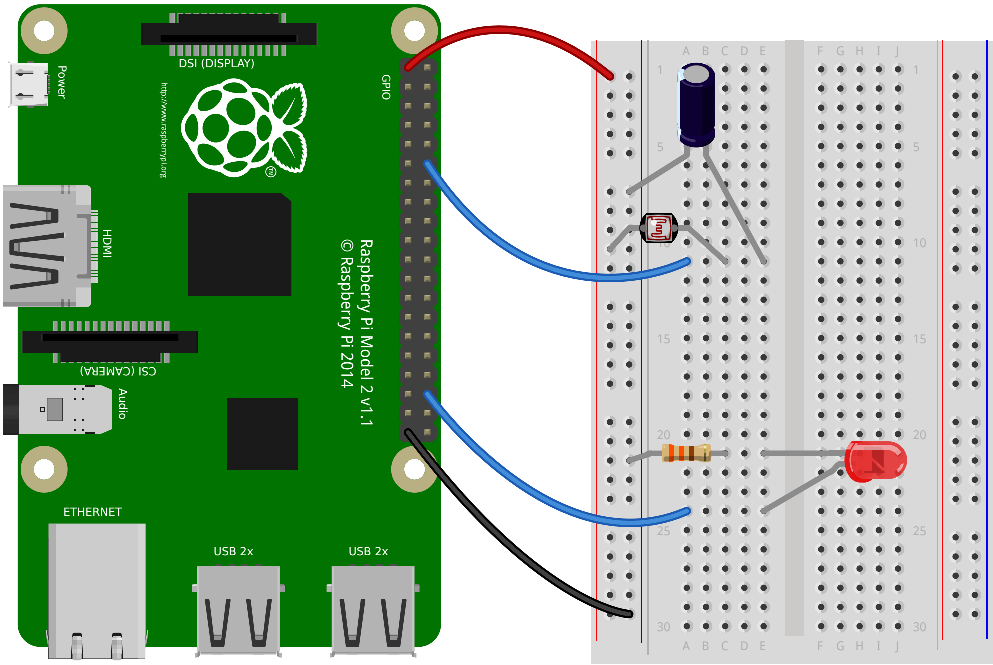
Motion sensor



Light an *LED* when a *MotionSensor* detects motion:

```
from gpiozero import MotionSensor, LED
from signal import pause

pir = MotionSensor(4)
led = LED(16)

pir.when_motion = led.on
pir.when_no_motion = led.off

pause()
```

Light sensor



Have a *LightSensor* detect light and dark:

```
from gpiozero import LightSensor

sensor = LightSensor(18)

while True:
    sensor.wait_for_light()
    print("It's light! :)")
    sensor.wait_for_dark()
    print("It's dark :(")
```

Run a function when the light changes:

```
from gpiozero import LightSensor, LED
from signal import pause

sensor = LightSensor(18)
led = LED(16)

sensor.when_dark = led.on
sensor.when_light = led.off

pause()
```

Or make a *PWMLED* change brightness according to the detected light level:

```
from gpiozero import LightSensor, PWMLED
from signal import pause

sensor = LightSensor(18)
led = PWMLED(16)

led.source = sensor.values

pause()
```

Distance sensor

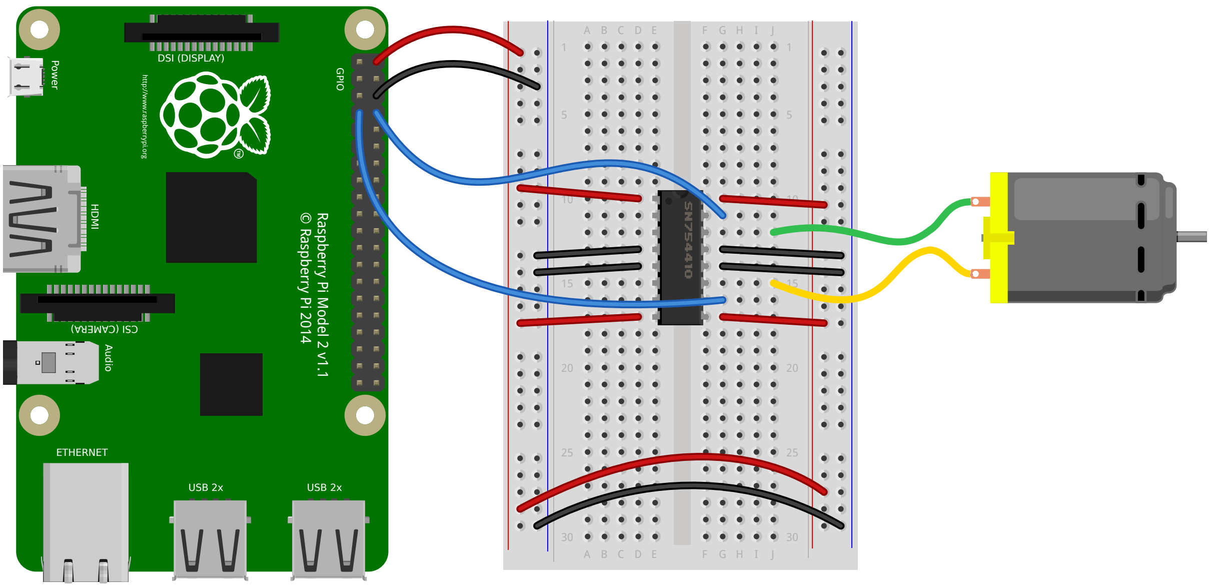
Have a *DistanceSensor* detect the distance to the nearest object:

```
from gpiozero import DistanceSensor
from time import sleep

sensor = DistanceSensor(23, 24)

while True:
    print('Distance to nearest object is', sensor.distance, 'm')
    sleep(1)
```

Run a function when something gets near the sensor:

```
from gpiozero import DistanceSensor, LED
from signal import pause

sensor = DistanceSensor(23, 24, max_distance=1, threshold_distance=0.2)
led = LED(16)

sensor.when_in_range = led.on
sensor.when_out_of_range = led.off

pause()
```

Motors



Spin a *Motor* around forwards and backwards:

```
from gpiozero import Motor
from time import sleep

motor = Motor(forward=4, backward=14)

while True:
    motor.forward()
    sleep(5)
    motor.backward()
    sleep(5)
```

Robot

Make a *Robot* drive around in (roughly) a square:

```
from gpiozero import Robot
from time import sleep

robot = Robot(left=(4, 14), right=(17, 18))

for i in range(4):
    robot.forward()
    sleep(10)
    robot.right()
    sleep(1)
```

Make a robot with a distance sensor that runs away when things get within 20cm of it:

```
from gpiozero import Robot, DistanceSensor
from signal import pause

sensor = DistanceSensor(23, 24, max_distance=1, threshold_distance=0.2)
robot = Robot(left=(4, 14), right=(17, 18))

sensor.when_in_range = robot.backward
sensor.when_out_of_range = robot.stop
pause()
```

Button controlled robot

Use four GPIO buttons as forward/back/left/right controls for a robot:

```
from gpiozero import Robot, Button
from signal import pause

robot = Robot(left=(4, 14), right=(17, 18))

left = Button(26)
right = Button(16)
fw = Button(21)
bw = Button(20)

fw.when_pressed = robot.forward
fw.when_released = robot.stop

left.when_pressed = robot.left
left.when_released = robot.stop

right.when_pressed = robot.right
right.when_released = robot.stop

bw.when_pressed = robot.backward
bw.when_released = robot.stop

pause()
```

Keyboard controlled robot

Use up/down/left/right keys to control a robot:

```
import curses
from gpiozero import Robot

robot = Robot(left=(4, 14), right=(17, 18))

actions = {
    curses.KEY_UP:    robot.forward,
    curses.KEY_DOWN:  robot.backward,
    curses.KEY_LEFT:  robot.left,
    curses.KEY_RIGHT: robot.right,
    }

def main(window):
    next_key = None
    while True:
        curses.halfdelay(1)
        if next_key is None:
            key = window.getch()
        else:
            key = next_key
            next_key = None
        if key != -1:
            # KEY DOWN
            curses.halfdelay(3)
            action = actions.get(key)
            if action is not None:
                action()
            next_key = key
            while next_key == key:
                next_key = window.getch()
            # KEY UP
            robot.stop()

curses.wrapper(main)
```

Note: This recipe uses the standard `curses` module. This module requires that Python is running in a terminal in order to work correctly, hence this recipe will *not* work in environments like IDLE.

If you prefer a version that works under IDLE, the following recipe should suffice:

```
from gpiozero import Robot
from evdev import InputDevice, list_devices, ecodes

robot = Robot(left=(4, 14), right=(17, 18))

# Get the list of available input devices
devices = [InputDevice(device) for device in list_devices()]
# Filter out everything that's not a keyboard. Keyboards are defined as any
# device which has keys, and which specifically has keys 1..31 (roughly Esc,
# the numeric keys, the first row of QWERTY plus a few more) and which does
# *not* have key 0 (reserved)
must_have = {i for i in range(1, 32)}
must_not_have = {0}
devices = [
    dev
    for dev in devices
    for keys in (set(dev.capabilities().get(ecodes.EV_KEY, [])),)
    if must_have.issubset(keys)
    and must_not_have.isdisjoint(keys)
]
# Pick the first keyboard
```

```
keyboard = devices[0]

keypress_actions = {
    ecodes.KEY_UP: robot.forward,
    ecodes.KEY_DOWN: robot.backward,
    ecodes.KEY_LEFT: robot.left,
    ecodes.KEY_RIGHT: robot.right,
}

for event in keyboard.read_loop():
    if event.type == ecodes.EV_KEY and event.code in keypress_actions:
        if event.value == 1:  # key down
            keypress_actions[event.code]()
        if event.value == 0:  # key up
            robot.stop()
```

Note: This recipe uses the third-party `evdev` module. Install this library with `sudo pip3 install evdev` first. Be aware that `evdev` will only work with local input devices; this recipe will *not* work over SSH.

Motion sensor robot

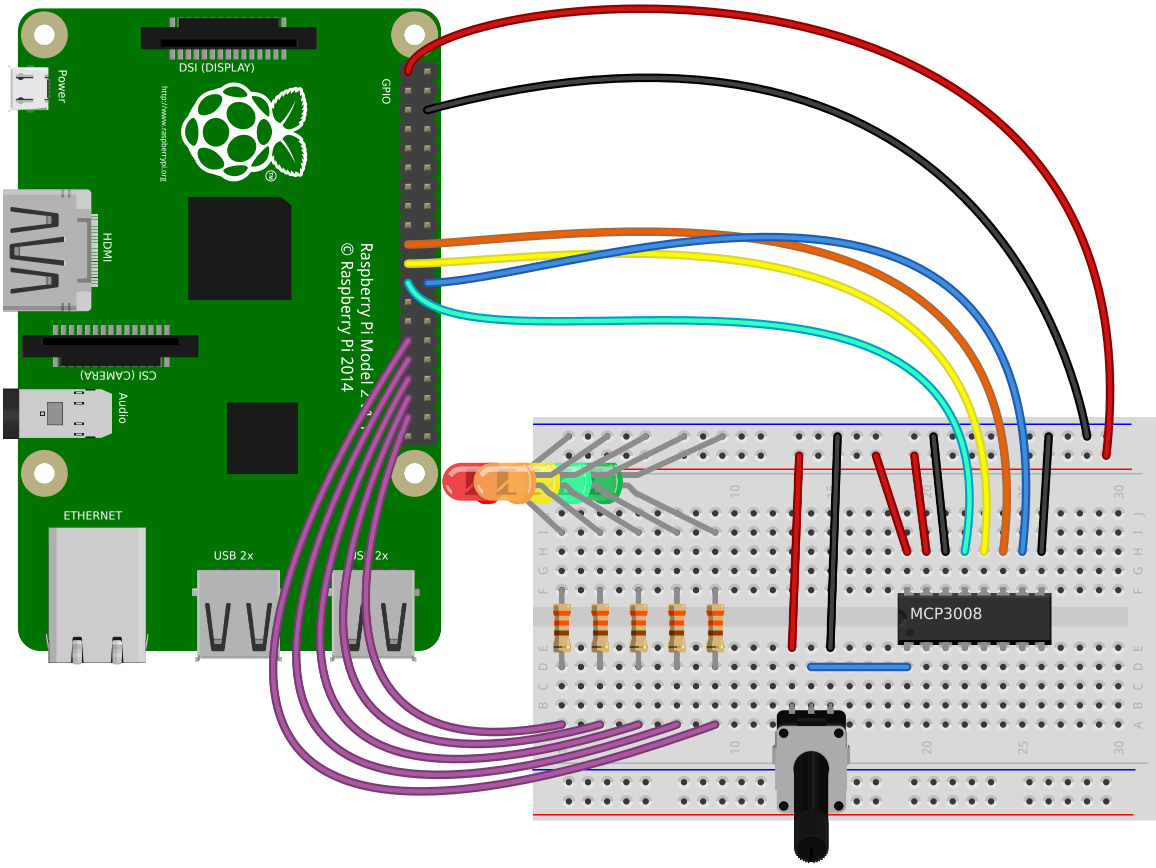
Make a robot drive forward when it detects motion:

```
from gpiozero import Robot, MotionSensor
from signal import pause

robot = Robot(left=(4, 14), right=(17, 18))
pir = MotionSensor(5)

pir.when_motion = robot.forward
pir.when_no_motion = robot.stop

pause()
```

Alternatively:

```
from gpiozero import Robot, MotionSensor
from signal import pause

robot = Robot(left=(4, 14), right=(17, 18))
pir = MotionSensor(5)

robot.source = zip(pir.values, pir.values)

pause()
```

Potentiometer



Continually print the value of a potentiometer (values between 0 and 1) connected to a *MCP3008* analog to digital converter:

```
from gpiozero import MCP3008

pot = MCP3008(channel=0)

while True:
    print(pot.value)
```

Present the value of a potentiometer on an LED bar graph using PWM to represent states that won't "fill" an LED:

```
from gpiozero import LEDBarGraph, MCP3008
from signal import pause

graph = LEDBarGraph(5, 6, 13, 19, 26, pwm=True)
pot = MCP3008(channel=0)
graph.source = pot.values
pause()
```

Measure temperature with an ADC

Wire a TMP36 temperature sensor to the first channel of an *MCP3008* analog to digital converter:

```
from gpiozero import MCP3008
from time import sleep

def convert_temp(gen):
```

```
    for value in gen:
        yield (value * 3.3 - 0.5) * 100

adc = MCP3008(channel=0)

for temp in convert_temp(adc.values):
    print('The temperature is', temp, 'C')
    sleep(1)
```

Full color LED controlled by 3 potentiometers

Wire up three potentiometers (for red, green and blue) and use each of their values to make up the colour of the LED:

```
from gpiozero import RGBLED, MCP3008

led = RGBLED(red=2, green=3, blue=4)
red_pot = MCP3008(channel=0)
green_pot = MCP3008(channel=1)
blue_pot = MCP3008(channel=2)

while True:
    led.red = red_pot.value
    led.green = green_pot.value
    led.blue = blue_pot.value
```

Alternatively, the following example is identical, but uses the `source` property rather than a `while` loop:

```
from gpiozero import RGBLED, MCP3008
from signal import pause

led = RGBLED(2, 3, 4)
red_pot = MCP3008(0)
green_pot = MCP3008(1)
blue_pot = MCP3008(2)

led.source = zip(red_pot.values, green_pot.values, blue_pot.values)

pause()
```

Please note the example above requires Python 3. In Python 2, `zip()` doesn't support lazy evaluation so the script will simply hang.

Controlling the Pi's own LEDs

On certain models of Pi (specifically the model A+, B+, and 2B) it's possible to control the power and activity LEDs. This can be useful for testing GPIO functionality without the need to wire up your own LEDs (also useful because the power and activity LEDs are "known good").

Firstly you need to disable the usual triggers for the built-in LEDs. This can be done from the terminal with the following commands:

```
$ echo none | sudo tee /sys/class/leds/led0/trigger
$ echo gpio | sudo tee /sys/class/leds/led1/trigger
```

Now you can control the LEDs with gpiozero like so:

```
from gpiozero import LED
from signal import pause
```

```
power = LED(35) # /sys/class/leds/led1
activity = LED(47) # /sys/class/leds/led0

activity.blink()
power.blink()
pause()
```

To revert the LEDs to their usual purpose you can either reboot your Pi or run the following commands:

```
$ echo mmc0 | sudo tee /sys/class/leds/led0/trigger
$ echo input | sudo tee /sys/class/leds/led1/trigger
```

Note: On the Pi Zero you can control the activity LED with this recipe, but there's no separate power LED to control (it's also worth noting the activity LED is active low, so set `active_high=False` when constructing your LED component).

On the original Pi 1 (model A or B), the activity LED can be controlled with GPIO16 (after disabling its trigger as above) but the power LED is hard-wired on.

On the Pi 3B the LEDs are controlled by a GPIO expander which is not accessible from gpiozero (yet).

Notes

Keep your script running

The following script looks like it should turn an LED on:

```
from gpiozero import LED

led = LED(17)
led.on()
```

And it does, if you're using the Python (or IPython or IDLE) shell. However, if you saved this script as a Python file and ran it, it would flash on briefly, then the script would end and it would turn off.

The following file includes an intentional `pause()` to keep the script alive:

```
from gpiozero import LED
from signal import pause

led = LED(17)
led.on()
pause()
```

Now the script will stay running, leaving the LED on, until it is terminated manually (e.g. by pressing Ctrl+C). Similarly, when setting up callbacks on button presses or other input devices, the script needs to be running for the events to be detected:

```
from gpiozero import Button
from signal import pause

def hello():
    print("Hello")

button = Button(2)
button.when_pressed = hello
pause()
```

Importing from GPIO Zero

In Python, libraries and functions used in a script must be imported by name at the top of the file, with the exception of the functions built into Python by default.

For example, to use the *Button* interface from GPIO Zero, it should be explicitly imported:

```
from gpiozero import Button
```

Now *Button* is available directly in your script:

```
button = Button(2)
```

Alternatively, the whole GPIO Zero library can be imported:

```
import gpiozero
```

In this case, all references to items within GPIO Zero must be prefixed:

```
button = gpiozero.Button(2)
```

How can I tell what version of gpiozero I have installed?

The gpiozero library relies on the setuptools package for installation services. You can use the setuptools `pkg_resources` API to query which version of gpiozero is available in your Python environment like so:

```
>>> from pkg_resources import require
>>> require('gpiozero')
[gpiozero 1.2.0 (/usr/local/lib/python2.7/dist-packages)]
>>> require('gpiozero')[0].version
'1.2.0'
```

If you have multiple versions installed (e.g. from `pip` and `apt-get`) they will not show up in the list returned by the `require` method. However, the first entry in the list will be the version that `import gpiozero` will import.

If you receive the error "No module named pkg_resources", you need to install the `pip` utility. This can be done with the following command in Raspbian:

```
$ sudo apt-get install python-pip
```

Contributing

This module was designed for use in education; particularly for young children. It is intended to provide a simple interface to everyday components.

If a proposed change added an advanced feature but made basic usage more complex, it is unlikely to be added.

Suggestions

Please make suggestions for additional components or enhancements to the codebase by opening an issue explaining your reasoning clearly.

Bugs

Please submit bug reports by opening an issue explaining the problem clearly using code examples.

Documentation

The documentation source lives in the docs folder. Contributions to the documentation are welcome but should be easy to read and understand.

Commit messages and pull requests

Commit messages should be concise but descriptive, and in the form of a patch description, i.e. instructional not past tense ("Add LED example" not "Added LED example"). Commits that close (or intend to close) an issue should use the phrase "fix #123" where `#123` is the issue number.

Backwards compatibility

Since this library reached v1.0 we aim to maintain backwards-compatibility thereafter. Changes which break backwards-compatibility will not be accepted.

Python

- Python 2/3 compatibility
- PEP8-compliance (with exceptions)

Input Devices

These input device component interfaces have been provided for simple use of everyday components. Components must be wired up correctly before use in code.

Note: All GPIO pin numbers use Broadcom (BCM) numbering. See the *Recipes* page for more information.

Button

class `gpiozero.`**`Button`**(*pin*, *pull_up=True*, *bounce_time=None*)

Extends *`DigitalInputDevice`* and represents a simple push button or switch.

Connect one side of the button to a ground pin, and the other to any GPIO pin. Alternatively, connect one side of the button to the 3V3 pin, and the other to any GPIO pin, then set *pull_up* to `False` in the *`Button`* constructor.

The following example will print a line of text when the button is pushed:

```
from gpiozero import Button

button = Button(4)
button.wait_for_press()
print("The button was pressed!")
```

Parameters

- **`pin`** (*`int`*) – The GPIO pin which the button is attached to. See *Pin Numbering* for valid pin numbers.
- **`pull_up`** (*`bool`*) – If `True` (the default), the GPIO pin will be pulled high by default. In this case, connect the other side of the button to ground. If `False`, the GPIO pin will be pulled low by default. In this case, connect the other side of the button to 3V3.

- **bounce_time** (*float*) – If None (the default), no software bounce compensation will be performed. Otherwise, this is the length of time (in seconds) that the component will ignore changes in state after an initial change.
- **hold_time** (*float*) – The length of time (in seconds) to wait after the button is pushed, until executing the *when_held* handler. Defaults to 1.
- **hold_repeat** (*bool*) – If True, the *when_held* handler will be repeatedly executed as long as the device remains active, every *hold_time* seconds. If False (the default) the *when_held* handler will be only be executed once per hold.

wait_for_press(*timeout=None*)
Pause the script until the device is activated, or the timeout is reached.

Parameters timeout (*float*) – Number of seconds to wait before proceeding. If this is None (the default), then wait indefinitely until the device is active.

wait_for_release(*timeout=None*)
Pause the script until the device is deactivated, or the timeout is reached.

Parameters timeout (*float*) – Number of seconds to wait before proceeding. If this is None (the default), then wait indefinitely until the device is inactive.

held_time
The length of time (in seconds) that the device has been held for. This is counted from the first execution of the *when_held* event rather than when the device activated, in contrast to *active_time*. If the device is not currently held, this is None.

hold_repeat
If True, *when_held* will be executed repeatedly with *hold_time* seconds between each invocation.

hold_time
The length of time (in seconds) to wait after the device is activated, until executing the *when_held* handler. If *hold_repeat* is True, this is also the length of time between invocations of *when_held*.

is_held
When True, the device has been active for at least *hold_time* seconds.

is_pressed
Returns True if the device is currently active and False otherwise. This property is usually derived from value. Unlike value, this is *always* a boolean.

pin
The *Pin* that the device is connected to. This will be None if the device has been closed (see the close() method). When dealing with GPIO pins, query pin.number to discover the GPIO pin (in BCM numbering) that the device is connected to.

pull_up
If True, the device uses a pull-up resistor to set the GPIO pin "high" by default.

when_held
The function to run when the device has remained active for *hold_time* seconds.

This can be set to a function which accepts no (mandatory) parameters, or a Python function which accepts a single mandatory parameter (with as many optional parameters as you like). If the function accepts a single mandatory parameter, the device that activated will be passed as that parameter.

Set this property to None (the default) to disable the event.

when_pressed
The function to run when the device changes state from inactive to active.

This can be set to a function which accepts no (mandatory) parameters, or a Python function which accepts a single mandatory parameter (with as many optional parameters as you like). If the function accepts a single mandatory parameter, the device that activated will be passed as that parameter.

Set this property to `None` (the default) to disable the event.

when_released
The function to run when the device changes state from active to inactive.

This can be set to a function which accepts no (mandatory) parameters, or a Python function which accepts a single mandatory parameter (with as many optional parameters as you like). If the function accepts a single mandatory parameter, the device that deactivated will be passed as that parameter.

Set this property to `None` (the default) to disable the event.

Line Sensor (TRCT5000)

class gpiozero.**LineSensor**(*pin*)
Extends *SmoothedInputDevice* and represents a single pin line sensor like the TCRT5000 infra-red proximity sensor found in the CamJam #3 EduKit.

A typical line sensor has a small circuit board with three pins: VCC, GND, and OUT. VCC should be connected to a 3V3 pin, GND to one of the ground pins, and finally OUT to the GPIO specified as the value of the *pin* parameter in the constructor.

The following code will print a line of text indicating when the sensor detects a line, or stops detecting a line:

```
from gpiozero import LineSensor
from signal import pause

sensor = LineSensor(4)
sensor.when_line = lambda: print('Line detected')
sensor.when_no_line = lambda: print('No line detected')
pause()
```

Parameters

- **pin** (*int*) – The GPIO pin which the sensor is attached to. See *Pin Numbering* for valid pin numbers.
- **queue_len** (*int*) – The length of the queue used to store values read from the sensor. This defaults to 5.
- **sample_rate** (*float*) – The number of values to read from the device (and append to the internal queue) per second. Defaults to 100.
- **threshold** (*float*) – Defaults to 0.5. When the mean of all values in the internal queue rises above this value, the sensor will be considered "active" by the *is_active* property, and all appropriate events will be fired.
- **partial** (*bool*) – When `False` (the default), the object will not return a value for *is_active* until the internal queue has filled with values. Only set this to `True` if you require values immediately after object construction.

wait_for_line(*timeout=None*)
Pause the script until the device is deactivated, or the timeout is reached.

Parameters **timeout** (*float*) – Number of seconds to wait before proceeding. If this is `None` (the default), then wait indefinitely until the device is inactive.

wait_for_no_line(*timeout=None*)
Pause the script until the device is activated, or the timeout is reached.

Parameters **timeout** (*float*) – Number of seconds to wait before proceeding. If this is `None` (the default), then wait indefinitely until the device is active.

pin
The *Pin* that the device is connected to. This will be None if the device has been closed (see the close() method). When dealing with GPIO pins, query pin.number to discover the GPIO pin (in BCM numbering) that the device is connected to.

when_line
The function to run when the device changes state from active to inactive.

This can be set to a function which accepts no (mandatory) parameters, or a Python function which accepts a single mandatory parameter (with as many optional parameters as you like). If the function accepts a single mandatory parameter, the device that deactivated will be passed as that parameter.

Set this property to None (the default) to disable the event.

when_no_line
The function to run when the device changes state from inactive to active.

This can be set to a function which accepts no (mandatory) parameters, or a Python function which accepts a single mandatory parameter (with as many optional parameters as you like). If the function accepts a single mandatory parameter, the device that activated will be passed as that parameter.

Set this property to None (the default) to disable the event.

Motion Sensor (D-SUN PIR)

class gpiozero.**MotionSensor**(*pin*, *queue_len=1*, *sample_rate=10*, *threshold=0.5*, *partial=False*)
Extends *SmoothedInputDevice* and represents a passive infra-red (PIR) motion sensor like the sort found in the CamJam #2 EduKit.

A typical PIR device has a small circuit board with three pins: VCC, OUT, and GND. VCC should be connected to a 5V pin, GND to one of the ground pins, and finally OUT to the GPIO specified as the value of the *pin* parameter in the constructor.

The following code will print a line of text when motion is detected:

```
from gpiozero import MotionSensor

pir = MotionSensor(4)
pir.wait_for_motion()
print("Motion detected!")
```

Parameters

- **pin** (*int*) – The GPIO pin which the sensor is attached to. See *Pin Numbering* for valid pin numbers.
- **queue_len** (*int*) – The length of the queue used to store values read from the sensor. This defaults to 1 which effectively disables the queue. If your motion sensor is particularly "twitchy" you may wish to increase this value.
- **sample_rate** (*float*) – The number of values to read from the device (and append to the internal queue) per second. Defaults to 100.
- **threshold** (*float*) – Defaults to 0.5. When the mean of all values in the internal queue rises above this value, the sensor will be considered "active" by the *is_active* property, and all appropriate events will be fired.
- **partial** (*bool*) – When False (the default), the object will not return a value for *is_active* until the internal queue has filled with values. Only set this to True if you require values immediately after object construction.

wait_for_motion(*timeout=None*)
Pause the script until the device is activated, or the timeout is reached.

Parameters **timeout** (*float*) – Number of seconds to wait before proceeding. If this is `None` (the default), then wait indefinitely until the device is active.

wait_for_no_motion (*timeout=None*)
Pause the script until the device is deactivated, or the timeout is reached.

Parameters **timeout** (*float*) – Number of seconds to wait before proceeding. If this is `None` (the default), then wait indefinitely until the device is inactive.

motion_detected
Returns `True` if the device is currently active and `False` otherwise.

pin
The *Pin* that the device is connected to. This will be `None` if the device has been closed (see the `close()` method). When dealing with GPIO pins, query `pin.number` to discover the GPIO pin (in BCM numbering) that the device is connected to.

when_motion
The function to run when the device changes state from inactive to active.

This can be set to a function which accepts no (mandatory) parameters, or a Python function which accepts a single mandatory parameter (with as many optional parameters as you like). If the function accepts a single mandatory parameter, the device that activated will be passed as that parameter.

Set this property to `None` (the default) to disable the event.

when_no_motion
The function to run when the device changes state from active to inactive.

This can be set to a function which accepts no (mandatory) parameters, or a Python function which accepts a single mandatory parameter (with as many optional parameters as you like). If the function accepts a single mandatory parameter, the device that deactivated will be passed as that parameter.

Set this property to `None` (the default) to disable the event.

Light Sensor (LDR)

class gpiozero.**LightSensor** (*pin, queue_len=5, charge_time_limit=0.01, threshold=0.1, partial=False*)
Extends *SmoothedInputDevice* and represents a light dependent resistor (LDR).

Connect one leg of the LDR to the 3V3 pin; connect one leg of a 1µF capacitor to a ground pin; connect the other leg of the LDR and the other leg of the capacitor to the same GPIO pin. This class repeatedly discharges the capacitor, then times the duration it takes to charge (which will vary according to the light falling on the LDR).

The following code will print a line of text when light is detected:

```
from gpiozero import LightSensor

ldr = LightSensor(18)
ldr.wait_for_light()
print("Light detected!")
```

Parameters

- **pin** (*int*) – The GPIO pin which the sensor is attached to. See *Pin Numbering* for valid pin numbers.
- **queue_len** (*int*) – The length of the queue used to store values read from the circuit. This defaults to 5.
- **charge_time_limit** (*float*) – If the capacitor in the circuit takes longer than this length of time to charge, it is assumed to be dark. The default (0.01 seconds) is

appropriate for a 1µF capacitor coupled with the LDR from the CamJam #2 EduKit. You may need to adjust this value for different valued capacitors or LDRs.

- **threshold** (*float*) – Defaults to 0.1. When the mean of all values in the internal queue rises above this value, the area will be considered "light", and all appropriate events will be fired.
- **partial** (*bool*) – When `False` (the default), the object will not return a value for *is_active* until the internal queue has filled with values. Only set this to `True` if you require values immediately after object construction.

wait_for_dark(*timeout=None*)
Pause the script until the device is deactivated, or the timeout is reached.

Parameters **timeout** (*float*) – Number of seconds to wait before proceeding. If this is `None` (the default), then wait indefinitely until the device is inactive.

wait_for_light(*timeout=None*)
Pause the script until the device is activated, or the timeout is reached.

Parameters **timeout** (*float*) – Number of seconds to wait before proceeding. If this is `None` (the default), then wait indefinitely until the device is active.

light_detected
Returns `True` if the device is currently active and `False` otherwise.

pin
The *Pin* that the device is connected to. This will be `None` if the device has been closed (see the `close()` method). When dealing with GPIO pins, query `pin.number` to discover the GPIO pin (in BCM numbering) that the device is connected to.

when_dark
The function to run when the device changes state from active to inactive.

This can be set to a function which accepts no (mandatory) parameters, or a Python function which accepts a single mandatory parameter (with as many optional parameters as you like). If the function accepts a single mandatory parameter, the device that deactivated will be passed as that parameter.

Set this property to `None` (the default) to disable the event.

when_light
The function to run when the device changes state from inactive to active.

This can be set to a function which accepts no (mandatory) parameters, or a Python function which accepts a single mandatory parameter (with as many optional parameters as you like). If the function accepts a single mandatory parameter, the device that activated will be passed as that parameter.

Set this property to `None` (the default) to disable the event.

Distance Sensor (HC-SR04)

class gpiozero.**DistanceSensor**(*echo*, *trigger*, *queue_len=30*, *max_distance=1*, *threshold_distance=0.3*, *partial=False*)
Extends *SmoothedInputDevice* and represents an HC-SR04 ultrasonic distance sensor, as found in the CamJam #3 EduKit.

The distance sensor requires two GPIO pins: one for the *trigger* (marked TRIG on the sensor) and another for the *echo* (marked ECHO on the sensor). However, a voltage divider is required to ensure the 5V from the ECHO pin doesn't damage the Pi. Wire your sensor according to the following instructions:

1. Connect the GND pin of the sensor to a ground pin on the Pi.
2. Connect the TRIG pin of the sensor a GPIO pin.
3. Connect a 330Ω resistor from the ECHO pin of the sensor to a different GPIO pin.

4.Connect a 470Ω resistor from ground to the ECHO GPIO pin. This forms the required voltage divider.

5.Finally, connect the VCC pin of the sensor to a 5V pin on the Pi.

The following code will periodically report the distance measured by the sensor in cm assuming the TRIG pin is connected to GPIO17, and the ECHO pin to GPIO18:

```
from gpiozero import DistanceSensor
from time import sleep

sensor = DistanceSensor(echo=18, trigger=17)
while True:
    print('Distance: ', sensor.distance * 100)
    sleep(1)
```

Parameters

- **echo** (*int*) – The GPIO pin which the ECHO pin is attached to. See *Pin Numbering* for valid pin numbers.
- **trigger** (*int*) – The GPIO pin which the TRIG pin is attached to. See *Pin Numbering* for valid pin numbers.
- **queue_len** (*int*) – The length of the queue used to store values read from the sensor. This defaults to 30.
- **max_distance** (*float*) – The `value` attribute reports a normalized value between 0 (too close to measure) and 1 (maximum distance). This parameter specifies the maximum distance expected in meters. This defaults to 1.
- **threshold_distance** (*float*) – Defaults to 0.3. This is the distance (in meters) that will trigger the `in_range` and `out_of_range` events when crossed.
- **partial** (*bool*) – When `False` (the default), the object will not return a value for *is_active* until the internal queue has filled with values. Only set this to `True` if you require values immediately after object construction.

wait_for_in_range (*timeout=None*)

Pause the script until the device is deactivated, or the timeout is reached.

Parameters timeout (*float*) – Number of seconds to wait before proceeding. If this is `None` (the default), then wait indefinitely until the device is inactive.

wait_for_out_of_range (*timeout=None*)

Pause the script until the device is activated, or the timeout is reached.

Parameters timeout (*float*) – Number of seconds to wait before proceeding. If this is `None` (the default), then wait indefinitely until the device is active.

distance

Returns the current distance measured by the sensor in meters. Note that this property will have a value between 0 and *max_distance*.

echo

Returns the *Pin* that the sensor's echo is connected to. This is simply an alias for the usual `pin` attribute.

max_distance

The maximum distance that the sensor will measure in meters. This value is specified in the constructor and is used to provide the scaling for the `value` attribute. When *distance* is equal to *max_distance*, `value` will be 1.

threshold_distance

The distance, measured in meters, that will trigger the *when_in_range* and *when_out_of_range* events when crossed. This is simply a meter-scaled variant of the usual `threshold` attribute.

trigger
Returns the *Pin* that the sensor's trigger is connected to.

when_in_range
The function to run when the device changes state from active to inactive.

This can be set to a function which accepts no (mandatory) parameters, or a Python function which accepts a single mandatory parameter (with as many optional parameters as you like). If the function accepts a single mandatory parameter, the device that deactivated will be passed as that parameter.

Set this property to `None` (the default) to disable the event.

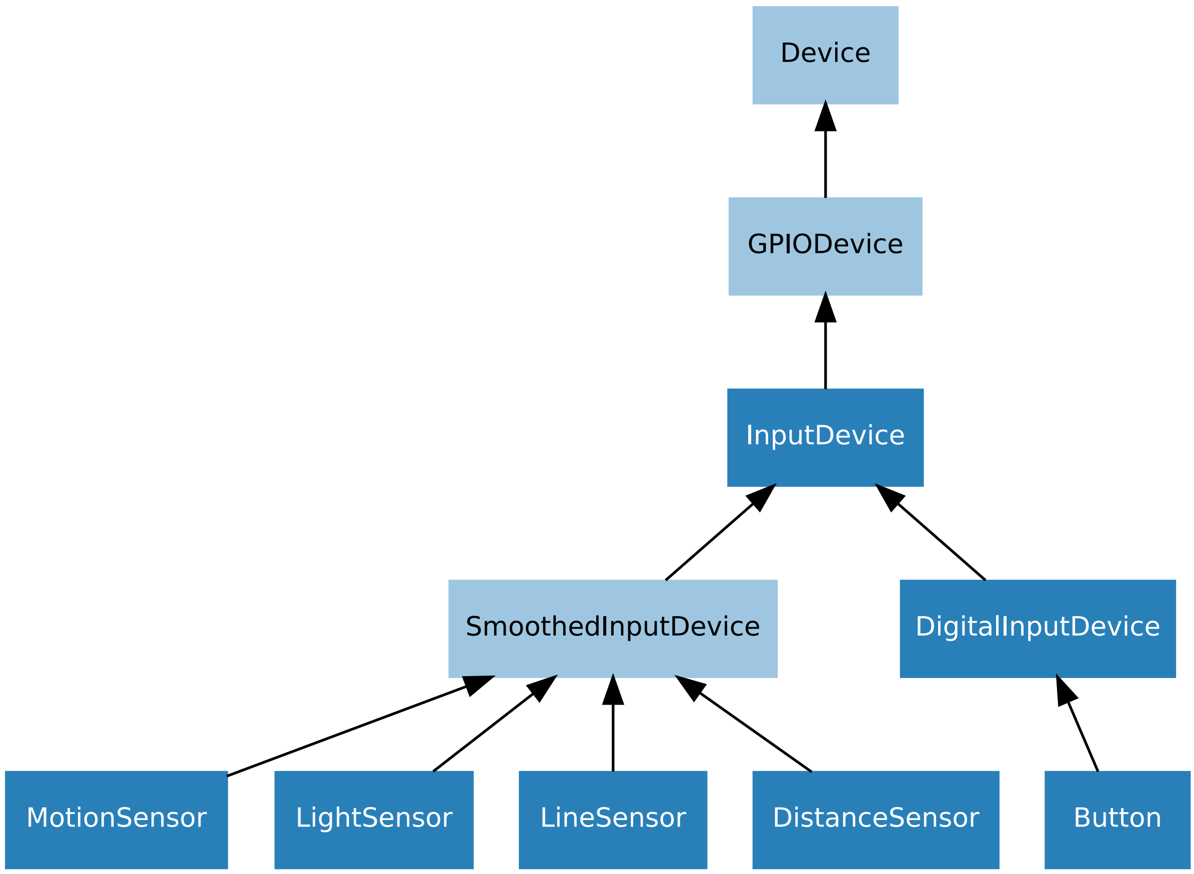
when_out_of_range
The function to run when the device changes state from inactive to active.

This can be set to a function which accepts no (mandatory) parameters, or a Python function which accepts a single mandatory parameter (with as many optional parameters as you like). If the function accepts a single mandatory parameter, the device that activated will be passed as that parameter.

Set this property to `None` (the default) to disable the event.

Base Classes

The classes in the sections above are derived from a series of base classes, some of which are effectively abstract. The classes form the (partial) hierarchy displayed in the graph below:



The following sections document these base classes for advanced users that wish to construct classes for their own devices.

DigitalInputDevice

class gpiozero.**DigitalInputDevice**(*pin*, *pull_up=False*, *bounce_time=None*)

Represents a generic input device with typical on/off behaviour.

This class extends *InputDevice* with machinery to fire the active and inactive events for devices that operate in a typical digital manner: straight forward on / off states with (reasonably) clean transitions between the two.

Parameters **bounce_time** (*float*) – Specifies the length of time (in seconds) that the component will ignore changes in state after an initial change. This defaults to None which indicates that no bounce compensation will be performed.

SmoothedInputDevice

class gpiozero.**SmoothedInputDevice**(*pin=None*, *pull_up=False*, *threshold=0.5*, *queue_len=5*, *sample_wait=0.0*, *partial=False*)

Represents a generic input device which takes its value from the mean of a queue of historical values.

This class extends *InputDevice* with a queue which is filled by a background thread which continually polls the state of the underlying device. The mean of the values in the queue is compared to a threshold which is used to determine the state of the *is_active* property.

Note: The background queue is not automatically started upon construction. This is to allow descendents to set up additional components before the queue starts reading values. Effectively this is an abstract base class.

This class is intended for use with devices which either exhibit analog behaviour (such as the charging time of a capacitor with an LDR), or those which exhibit "twitchy" behaviour (such as certain motion sensors).

Parameters

- **threshold** (*float*) – The value above which the device will be considered "on".
- **queue_len** (*int*) – The length of the internal queue which is filled by the background thread.
- **sample_wait** (*float*) – The length of time to wait between retrieving the state of the underlying device. Defaults to 0.0 indicating that values are retrieved as fast as possible.
- **partial** (*bool*) – If False (the default), attempts to read the state of the device (from the *is_active* property) will block until the queue has filled. If True, a value will be returned immediately, but be aware that this value is likely to fluctuate excessively.

close()

Shut down the device and release all associated resources. This method can be called on an already closed device without raising an exception.

This method is primarily intended for interactive use at the command line. It disables the device and releases its pin(s) for use by another device.

You can attempt to do this simply by deleting an object, but unless you've cleaned up all references to the object this may not work (even if you've cleaned up all references, there's still no guarantee the garbage collector will actually delete the object at that point). By contrast, the close method provides a means of ensuring that the object is shut down.

For example, if you have a breadboard with a buzzer connected to pin 16, but then wish to attach an LED instead:

```
>>> from gpiozero import *
>>> bz = Buzzer(16)
>>> bz.on()
>>> bz.off()
>>> bz.close()
>>> led = LED(16)
>>> led.blink()
```

`Device` descendents can also be used as context managers using the `with` statement. For example:

```
>>> from gpiozero import *
>>> with Buzzer(16) as bz:
...     bz.on()
...
>>> with LED(16) as led:
...     led.on()
...
```

is_active
Returns `True` if the device is currently active and `False` otherwise.

partial
If `False` (the default), attempts to read the *value* or *is_active* properties will block until the queue has filled.

queue_len
The length of the internal queue of values which is averaged to determine the overall state of the device. This defaults to 5.

threshold
If *value* exceeds this amount, then *is_active* will return `True`.

value
Returns the mean of the values in the internal queue. This is compared to *threshold* to determine whether *is_active* is `True`.

InputDevice

class gpiozero.**InputDevice**(*pin*, *pull_up=False*)
Represents a generic GPIO input device.

This class extends *GPIODevice* to add facilities common to GPIO input devices. The constructor adds the optional *pull_up* parameter to specify how the pin should be pulled by the internal resistors. The `is_active` property is adjusted accordingly so that `True` still means active regardless of the *pull_up* setting.

Parameters

- **pin** (*int*) – The GPIO pin (in Broadcom numbering) that the device is connected to. If this is `None` a *GPIODeviceError* will be raised.
- **pull_up** (*bool*) – If `True`, the pin will be pulled high with an internal resistor. If `False` (the default), the pin will be pulled low.

pull_up
If `True`, the device uses a pull-up resistor to set the GPIO pin "high" by default.

GPIODevice

class gpiozero.**GPIODevice**(*pin*)
Extends *Device*. Represents a generic GPIO device and provides the services common to all single-pin GPIO devices (like ensuring two GPIO devices do no share a *pin*).

Parameters **pin** (*int*) – The GPIO pin (in BCM numbering) that the device is connected to. If this is None, *GPIOPinMissing* will be raised. If the pin is already in use by another device, *GPIOPinInUse* will be raised.

close()

Shut down the device and release all associated resources. This method can be called on an already closed device without raising an exception.

This method is primarily intended for interactive use at the command line. It disables the device and releases its pin(s) for use by another device.

You can attempt to do this simply by deleting an object, but unless you've cleaned up all references to the object this may not work (even if you've cleaned up all references, there's still no guarantee the garbage collector will actually delete the object at that point). By contrast, the close method provides a means of ensuring that the object is shut down.

For example, if you have a breadboard with a buzzer connected to pin 16, but then wish to attach an LED instead:

```
>>> from gpiozero import *
>>> bz = Buzzer(16)
>>> bz.on()
>>> bz.off()
>>> bz.close()
>>> led = LED(16)
>>> led.blink()
```

Device descendents can also be used as context managers using the with statement. For example:

```
>>> from gpiozero import *
>>> with Buzzer(16) as bz:
...     bz.on()
...
>>> with LED(16) as led:
...     led.on()
...
```

pin

The *Pin* that the device is connected to. This will be None if the device has been closed (see the *close()* method). When dealing with GPIO pins, query pin.number to discover the GPIO pin (in BCM numbering) that the device is connected to.

Output Devices

These output device component interfaces have been provided for simple use of everyday components. Components must be wired up correctly before use in code.

Note: All GPIO pin numbers use Broadcom (BCM) numbering. See the *Recipes* page for more information.

LED

class gpiozero.**LED**(*pin*, *active_high=True*, *initial_value=False*)

Extends *DigitalOutputDevice* and represents a light emitting diode (LED).

Connect the cathode (short leg, flat side) of the LED to a ground pin; connect the anode (longer leg) to a limiting resistor; connect the other side of the limiting resistor to a GPIO pin (the limiting resistor can be placed either side of the LED).

The following example will light the LED:

```
from gpiozero import LED

led = LED(17)
led.on()
```

Parameters

- **pin** (*int*) – The GPIO pin which the LED is attached to. See *Pin Numbering* for valid pin numbers.
- **active_high** (*bool*) – If True (the default), the LED will operate normally with the circuit described above. If False you should wire the cathode to the GPIO pin, and the anode to a 3V3 pin (via a limiting resistor).
- **initial_value** (*bool*) – If False (the default), the LED will be off initially. If None, the LED will be left in whatever state the pin is found in when configured for output (warning: this can be on). If True, the LED will be switched on initially.

blink(*on_time=1*, *off_time=1*, *n=None*, *background=True*)
Make the device turn on and off repeatedly.

Parameters

- **on_time** (*float*) – Number of seconds on. Defaults to 1 second.
- **off_time** (*float*) – Number of seconds off. Defaults to 1 second.
- **n** (*int*) – Number of times to blink; None (the default) means forever.
- **background** (*bool*) – If True (the default), start a background thread to continue blinking and return immediately. If False, only return when the blink is finished (warning: the default value of *n* will result in this method never returning).

off()
Turns the device off.

on()
Turns the device on.

toggle()
Reverse the state of the device. If it's on, turn it off; if it's off, turn it on.

is_lit
Returns True if the device is currently active and False otherwise. This property is usually derived from value. Unlike value, this is *always* a boolean.

pin
The *Pin* that the device is connected to. This will be None if the device has been closed (see the close() method). When dealing with GPIO pins, query pin.number to discover the GPIO pin (in BCM numbering) that the device is connected to.

PWMLED

class gpiozero.**PWMLED**(*pin*, *active_high=True*, *initial_value=0*, *frequency=100*)
Extends *PWMOutputDevice* and represents a light emitting diode (LED) with variable brightness.

A typical configuration of such a device is to connect a GPIO pin to the anode (long leg) of the LED, and the cathode (short leg) to ground, with an optional resistor to prevent the LED from burning out.

Parameters

- **pin** (*int*) – The GPIO pin which the LED is attached to. See *Pin Numbering* for valid pin numbers.

- **active_high** (*bool*) – If True (the default), the *on()* method will set the GPIO to HIGH. If False, the *on()* method will set the GPIO to LOW (the *off()* method always does the opposite).
- **initial_value** (*float*) – If 0 (the default), the LED will be off initially. Other values between 0 and 1 can be specified as an initial brightness for the LED. Note that None cannot be specified (unlike the parent class) as there is no way to tell PWM not to alter the state of the pin.
- **frequency** (*int*) – The frequency (in Hz) of pulses emitted to drive the LED. Defaults to 100Hz.

blink (*on_time=1*, *off_time=1*, *fade_in_time=0*, *fade_out_time=0*, *n=None*, *background=True*)
Make the device turn on and off repeatedly.

Parameters

- **on_time** (*float*) – Number of seconds on. Defaults to 1 second.
- **off_time** (*float*) – Number of seconds off. Defaults to 1 second.
- **fade_in_time** (*float*) – Number of seconds to spend fading in. Defaults to 0.
- **fade_out_time** (*float*) – Number of seconds to spend fading out. Defaults to 0.
- **n** (*int*) – Number of times to blink; None (the default) means forever.
- **background** (*bool*) – If True (the default), start a background thread to continue blinking and return immediately. If False, only return when the blink is finished (warning: the default value of *n* will result in this method never returning).

off ()
Turns the device off.

on ()
Turns the device on.

pulse (*fade_in_time=1*, *fade_out_time=1*, *n=None*, *background=True*)
Make the device fade in and out repeatedly.

Parameters

- **fade_in_time** (*float*) – Number of seconds to spend fading in. Defaults to 1.
- **fade_out_time** (*float*) – Number of seconds to spend fading out. Defaults to 1.
- **n** (*int*) – Number of times to pulse; None (the default) means forever.
- **background** (*bool*) – If True (the default), start a background thread to continue pulsing and return immediately. If False, only return when the pulse is finished (warning: the default value of *n* will result in this method never returning).

toggle ()
Toggle the state of the device. If the device is currently off (*value* is 0.0), this changes it to "fully" on (*value* is 1.0). If the device has a duty cycle (*value*) of 0.1, this will toggle it to 0.9, and so on.

is_lit
Returns True if the device is currently active (*value* is non-zero) and False otherwise.

pin
The *Pin* that the device is connected to. This will be None if the device has been closed (see the close() method). When dealing with GPIO pins, query pin.number to discover the GPIO pin (in BCM numbering) that the device is connected to.

value
The duty cycle of the PWM device. 0.0 is off, 1.0 is fully on. Values in between may be specified for varying levels of power in the device.

RGBLED

class gpiozero.**RGBLED**(*red*, *green*, *blue*, *active_high=True*, *initial_value=(0, 0, 0)*, *pwm=True*)

Extends *Device* and represents a full color LED component (composed of red, green, and blue LEDs).

Connect the common cathode (longest leg) to a ground pin; connect each of the other legs (representing the red, green, and blue anodes) to any GPIO pins. You can either use three limiting resistors (one per anode) or a single limiting resistor on the cathode.

The following code will make the LED purple:

```
from gpiozero import RGBLED

led = RGBLED(2, 3, 4)
led.color = (1, 0, 1)
```

Parameters

- **red** (*int*) – The GPIO pin that controls the red component of the RGB LED.
- **green** (*int*) – The GPIO pin that controls the green component of the RGB LED.
- **blue** (*int*) – The GPIO pin that controls the blue component of the RGB LED.
- **active_high** (*bool*) – Set to `True` (the default) for common cathode RGB LEDs. If you are using a common anode RGB LED, set this to `False`.
- **initial_value** (*tuple*) – The initial color for the RGB LED. Defaults to black `(0, 0, 0)`.
- **pwm** (*bool*) – If `True` (the default), construct *PWMLED* instances for each component of the RGBLED. If `False`, construct regular *LED* instances, which prevents smooth color graduations.

blink(*on_time=1*, *off_time=1*, *fade_in_time=0*, *fade_out_time=0*, *on_color=(1, 1, 1)*, *off_color=(0, 0, 0)*, *n=None*, *background=True*)

Make the device turn on and off repeatedly.

Parameters

- **on_time** (*float*) – Number of seconds on. Defaults to 1 second.
- **off_time** (*float*) – Number of seconds off. Defaults to 1 second.
- **fade_in_time** (*float*) – Number of seconds to spend fading in. Defaults to 0. Must be 0 if `pwm` was `False` when the class was constructed (`ValueError` will be raised if not).
- **fade_out_time** (*float*) – Number of seconds to spend fading out. Defaults to 0. Must be 0 if `pwm` was `False` when the class was constructed (`ValueError` will be raised if not).
- **on_color** (*tuple*) – The color to use when the LED is "on". Defaults to white.
- **off_color** (*tuple*) – The color to use when the LED is "off". Defaults to black.
- **n** (*int*) – Number of times to blink; `None` (the default) means forever.
- **background** (*bool*) – If `True` (the default), start a background thread to continue blinking and return immediately. If `False`, only return when the blink is finished (warning: the default value of *n* will result in this method never returning).

off()

Turn the LED off. This is equivalent to setting the LED color to black `(0, 0, 0)`.

on()

Turn the LED on. This equivalent to setting the LED color to white `(1, 1, 1)`.

pulse (*fade_in_time=1*, *fade_out_time=1*, *on_color=(1, 1, 1)*, *off_color=(0, 0, 0)*, *n=None*, *background=True*)
Make the device fade in and out repeatedly.

Parameters

- **fade_in_time** (*float*) – Number of seconds to spend fading in. Defaults to 1.
- **fade_out_time** (*float*) – Number of seconds to spend fading out. Defaults to 1.
- **on_color** (*tuple*) – The color to use when the LED is "on". Defaults to white.
- **off_color** (*tuple*) – The color to use when the LED is "off". Defaults to black.
- **n** (*int*) – Number of times to pulse; `None` (the default) means forever.
- **background** (*bool*) – If `True` (the default), start a background thread to continue pulsing and return immediately. If `False`, only return when the pulse is finished (warning: the default value of *n* will result in this method never returning).

toggle ()
Toggle the state of the device. If the device is currently off (`value` is `(0, 0, 0)`), this changes it to "fully" on (`value` is `(1, 1, 1)`). If the device has a specific color, this method inverts the color.

color
Represents the color of the LED as an RGB 3-tuple of `(red, green, blue)` where each value is between 0 and 1 if `pwm` was `True` when the class was constructed (and only 0 or 1 if not).

For example, purple would be `(1, 0, 1)` and yellow would be `(1, 1, 0)`, while orange would be `(1, 0.5, 0)`.

is_lit
Returns `True` if the LED is currently active (not black) and `False` otherwise.

Buzzer

class gpiozero.**Buzzer** (*pin*, *active_high=True*, *initial_value=False*)
Extends `DigitalOutputDevice` and represents a digital buzzer component.

Connect the cathode (negative pin) of the buzzer to a ground pin; connect the other side to any GPIO pin.

The following example will sound the buzzer:

```
from gpiozero import Buzzer

bz = Buzzer(3)
bz.on()
```

Parameters

- **pin** (*int*) – The GPIO pin which the buzzer is attached to. See *Pin Numbering* for valid pin numbers.
- **active_high** (*bool*) – If `True` (the default), the buzzer will operate normally with the circuit described above. If `False` you should wire the cathode to the GPIO pin, and the anode to a 3V3 pin.
- **initial_value** (*bool*) – If `False` (the default), the buzzer will be silent initially. If `None`, the buzzer will be left in whatever state the pin is found in when configured for output (warning: this can be on). If `True`, the buzzer will be switched on initially.

beep (*on_time=1*, *off_time=1*, *n=None*, *background=True*)
Make the device turn on and off repeatedly.

Parameters

- **on_time** (*float*) – Number of seconds on. Defaults to 1 second.
- **off_time** (*float*) – Number of seconds off. Defaults to 1 second.
- **n** (*int*) – Number of times to blink; None (the default) means forever.
- **background** (*bool*) – If True (the default), start a background thread to continue blinking and return immediately. If False, only return when the blink is finished (warning: the default value of *n* will result in this method never returning).

off()
Turns the device off.

on()
Turns the device on.

toggle()
Reverse the state of the device. If it's on, turn it off; if it's off, turn it on.

is_active
Returns True if the device is currently active and False otherwise. This property is usually derived from value. Unlike value, this is *always* a boolean.

pin
The *Pin* that the device is connected to. This will be None if the device has been closed (see the close() method). When dealing with GPIO pins, query pin.number to discover the GPIO pin (in BCM numbering) that the device is connected to.

Motor

class gpiozero.**Motor**(*forward*, *backward*, *pwm=True*)
Extends *CompositeDevice* and represents a generic motor connected to a bi-directional motor driver circuit (i.e. an H-bridge).

Attach an H-bridge motor controller to your Pi; connect a power source (e.g. a battery pack or the 5V pin) to the controller; connect the outputs of the controller board to the two terminals of the motor; connect the inputs of the controller board to two GPIO pins.

The following code will make the motor turn "forwards":

```
from gpiozero import Motor

motor = Motor(17, 18)
motor.forward()
```

Parameters

- **forward** (*int*) – The GPIO pin that the forward input of the motor driver chip is connected to.
- **backward** (*int*) – The GPIO pin that the backward input of the motor driver chip is connected to.
- **pwm** (*bool*) – If True (the default), construct *PWMOutputDevice* instances for the motor controller pins, allowing both direction and variable speed control. If False, construct *DigitalOutputDevice* instances, allowing only direction control.

backward(*speed=1*)
Drive the motor backwards.

Parameters speed (*float*) – The speed at which the motor should turn. Can be any value between 0 (stopped) and the default 1 (maximum speed) if pwm was True when the class was constructed (and only 0 or 1 if not).

forward(*speed=1*)
Drive the motor forwards.

Parameters **speed** (*float*) – The speed at which the motor should turn. Can be any value between 0 (stopped) and the default 1 (maximum speed) if `pwm` was `True` when the class was constructed (and only 0 or 1 if not).

stop()
Stop the motor.

Servo

class gpiozero.**Servo**(*pin*, *initial_value=0*, *min_pulse_width=1/1000*, *max_pulse_width=2/1000*, *frame_width=20/1000*)
Extends `CompositeDevice` and represents a PWM-controlled servo motor connected to a GPIO pin.

Connect a power source (e.g. a battery pack or the 5V pin) to the power cable of the servo (this is typically colored red); connect the ground cable of the servo (typically colored black or brown) to the negative of your battery pack, or a GND pin; connect the final cable (typically colored white or orange) to the GPIO pin you wish to use for controlling the servo.

The following code will make the servo move between its minimum, maximum, and mid-point positions with a pause between each:

```
from gpiozero import Servo
from time import sleep

servo = Servo(17)
while True:
    servo.min()
    sleep(1)
    servo.mid()
    sleep(1)
    servo.max()
    sleep(1)
```

Parameters

- **pin** (*int*) – The GPIO pin which the device is attached to. See *Pin Numbering* for valid pin numbers.
- **initial_value** (*float*) – If `0` (the default), the device's mid-point will be set initially. Other values between -1 and +1 can be specified as an initial position. `None` means to start the servo un-controlled (see `value`).
- **min_pulse_width** (*float*) – The pulse width corresponding to the servo's minimum position. This defaults to 1ms.
- **max_pulse_width** (*float*) – The pulse width corresponding to the servo's maximum position. This defaults to 2ms.
- **frame_width** (*float*) – The length of time between servo control pulses measured in seconds. This defaults to 20ms which is a common value for servos.

detach()
Temporarily disable control of the servo. This is equivalent to setting `value` to `None`.

max()
Set the servo to its maximum position.

mid()
Set the servo to its mid-point position.

min()
Set the servo to its minimum position.

frame_width
The time between control pulses, measured in seconds.

max_pulse_width
The control pulse width corresponding to the servo's maximum position, measured in seconds.

min_pulse_width
The control pulse width corresponding to the servo's minimum position, measured in seconds.

pulse_width
Returns the current pulse width controlling the servo.

source
The iterable to use as a source of values for `value`.

source_delay
The delay (measured in seconds) in the loop used to read values from `source`. Defaults to 0.01 seconds which is generally sufficient to keep CPU usage to a minimum while providing adequate responsiveness.

value
Represents the position of the servo as a value between -1 (the minimum position) and +1 (the maximum position). This can also be the special value `None` indicating that the servo is currently "uncontrolled", i.e. that no control signal is being sent. Typically this means the servo's position remains unchanged, but that it can be moved by hand.

values
An infinite iterator of values read from *value*.

AngularServo

class gpiozero.**AngularServo**(*pin*, *initial_angle=0*, *min_angle=-90*, *max_angle=90*, *min_pulse_width=1/1000*, *max_pulse_width=2/1000*, *frame_width=20/1000*)
Extends `Servo` and represents a rotational PWM-controlled servo motor which can be set to particular angles (assuming valid minimum and maximum angles are provided to the constructor).

Connect a power source (e.g. a battery pack or the 5V pin) to the power cable of the servo (this is typically colored red); connect the ground cable of the servo (typically colored black or brown) to the negative of your battery pack, or a GND pin; connect the final cable (typically colored white or orange) to the GPIO pin you wish to use for controlling the servo.

Next, calibrate the angles that the servo can rotate to. In an interactive Python session, construct a `Servo` instance. The servo should move to its mid-point by default. Set the servo to its minimum value, and measure the angle from the mid-point. Set the servo to its maximum value, and again measure the angle:

```
>>> from gpiozero import Servo
>>> s = Servo(17)
>>> s.min() # measure the angle
>>> s.max() # measure the angle
```

You should now be able to construct an `AngularServo` instance with the correct bounds:

```
>>> from gpiozero import AngularServo
>>> s = AngularServo(17, min_angle=-42, max_angle=44)
>>> s.angle = 0.0
>>> s.angle
0.0
>>> s.angle = 15
>>> s.angle
15.0
```

Note: You can set *min_angle* greater than *max_angle* if you wish to reverse the sense of the angles (e.g. `min_angle=45, max_angle=-45`). This can be useful with servos that rotate in the opposite direction to your expectations of minimum and maximum.

Parameters

- **pin** (*int*) – The GPIO pin which the device is attached to. See *Pin Numbering* for valid pin numbers.
- **initial_angle** (*float*) – Sets the servo's initial angle to the specified value. The default is 0. The value specified must be between *min_angle* and *max_angle* inclusive. `None` means to start the servo un-controlled (see `value`).
- **min_angle** (*float*) – Sets the minimum angle that the servo can rotate to. This defaults to -90, but should be set to whatever you measure from your servo during calibration.
- **max_angle** (*float*) – Sets the maximum angle that the servo can rotate to. This defaults to 90, but should be set to whatever you measure from your servo during calibration.
- **min_pulse_width** (*float*) – The pulse width corresponding to the servo's minimum position. This defaults to 1ms.
- **max_pulse_width** (*float*) – The pulse width corresponding to the servo's maximum position. This defaults to 2ms.
- **frame_width** (*float*) – The length of time between servo control pulses measured in seconds. This defaults to 20ms which is a common value for servos.

detach()
Temporarily disable control of the servo. This is equivalent to setting `value` to `None`.

max()
Set the servo to its maximum position.

mid()
Set the servo to its mid-point position.

min()
Set the servo to its minimum position.

angle
The position of the servo as an angle measured in degrees. This will only be accurate if *min_angle* and *max_angle* have been set appropriately in the constructor.

This can also be the special value `None` indicating that the servo is currently "uncontrolled", i.e. that no control signal is being sent. Typically this means the servo's position remains unchanged, but that it can be moved by hand.

frame_width
The time between control pulses, measured in seconds.

max_angle
The maximum angle that the servo will rotate to when `max()` is called.

max_pulse_width
The control pulse width corresponding to the servo's maximum position, measured in seconds.

min_angle
The minimum angle that the servo will rotate to when `min()` is called.

min_pulse_width
The control pulse width corresponding to the servo's minimum position, measured in seconds.

pulse_width
Returns the current pulse width controlling the servo.

source
The iterable to use as a source of values for *value*.

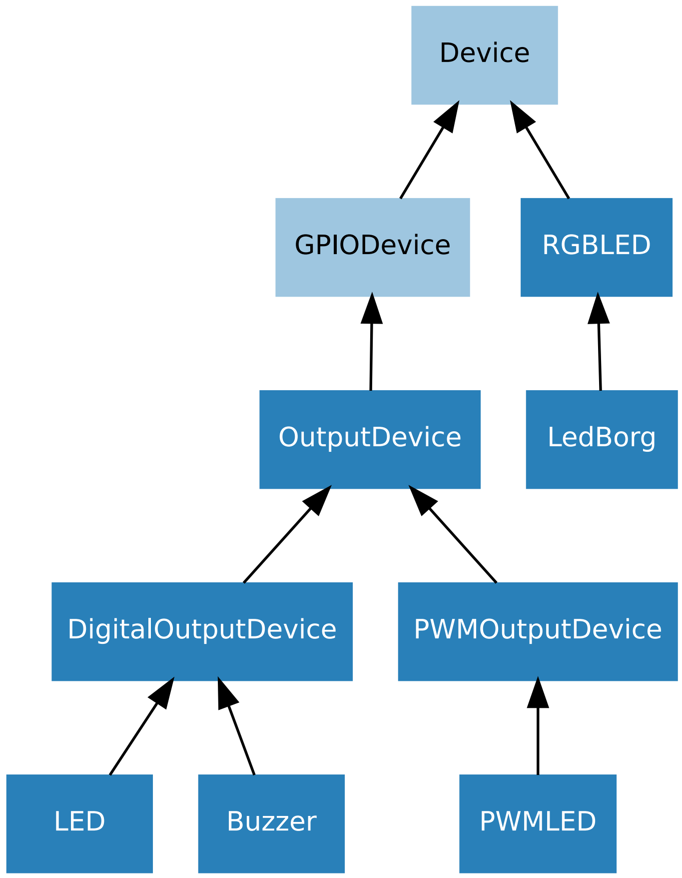
source_delay
The delay (measured in seconds) in the loop used to read values from *source*. Defaults to 0.01 seconds which is generally sufficient to keep CPU usage to a minimum while providing adequate responsiveness.

value
Represents the position of the servo as a value between -1 (the minimum position) and +1 (the maximum position). This can also be the special value `None` indicating that the servo is currently "uncontrolled", i.e. that no control signal is being sent. Typically this means the servo's position remains unchanged, but that it can be moved by hand.

values
An infinite iterator of values read from *value*.

Base Classes

The classes in the sections above are derived from a series of base classes, some of which are effectively abstract. The classes form the (partial) hierarchy displayed in the graph below:



The following sections document these base classes for advanced users that wish to construct classes for their own devices.

DigitalOutputDevice

class gpiozero.**DigitalOutputDevice**(*pin*, *active_high=True*, *initial_value=False*)

Represents a generic output device with typical on/off behaviour.

This class extends `OutputDevice` with a `blink()` method which uses an optional background thread to handle toggling the device state without further interaction.

blink(*on_time=1*, *off_time=1*, *n=None*, *background=True*)

Make the device turn on and off repeatedly.

Parameters

- **on_time** (*float*) – Number of seconds on. Defaults to 1 second.
- **off_time** (*float*) – Number of seconds off. Defaults to 1 second.
- **n** (*int*) – Number of times to blink; `None` (the default) means forever.
- **background** (*bool*) – If `True` (the default), start a background thread to continue blinking and return immediately. If `False`, only return when the blink is finished (warning: the default value of *n* will result in this method never returning).

close()

Shut down the device and release all associated resources. This method can be called on an already closed device without raising an exception.

This method is primarily intended for interactive use at the command line. It disables the device and releases its pin(s) for use by another device.

You can attempt to do this simply by deleting an object, but unless you've cleaned up all references to the object this may not work (even if you've cleaned up all references, there's still no guarantee the garbage collector will actually delete the object at that point). By contrast, the close method provides a means of ensuring that the object is shut down.

For example, if you have a breadboard with a buzzer connected to pin 16, but then wish to attach an LED instead:

```
>>> from gpiozero import *
>>> bz = Buzzer(16)
>>> bz.on()
>>> bz.off()
>>> bz.close()
>>> led = LED(16)
>>> led.blink()
```

`Device` descendents can also be used as context managers using the `with` statement. For example:

```
>>> from gpiozero import *
>>> with Buzzer(16) as bz:
...     bz.on()
...
>>> with LED(16) as led:
...     led.on()
...
```

off()

Turns the device off.

on()

Turns the device on.

PWMOutputDevice

class gpiozero.**PWMOutputDevice**(*pin*, *active_high=True*, *initial_value=0*, *frequency=100*)
Generic output device configured for pulse-width modulation (PWM).

Parameters

- **pin** (*int*) – The GPIO pin which the device is attached to. See *Pin Numbering* for valid pin numbers.
- **active_high** (*bool*) – If `True` (the default), the `on()` method will set the GPIO to HIGH. If `False`, the `on()` method will set the GPIO to LOW (the `off()` method always does the opposite).
- **initial_value** (*float*) – If `0` (the default), the device's duty cycle will be 0 initially. Other values between 0 and 1 can be specified as an initial duty cycle. Note that `None` cannot be specified (unlike the parent class) as there is no way to tell PWM not to alter the state of the pin.
- **frequency** (*int*) – The frequency (in Hz) of pulses emitted to drive the device. Defaults to 100Hz.

blink(*on_time=1*, *off_time=1*, *fade_in_time=0*, *fade_out_time=0*, *n=None*, *background=True*)
Make the device turn on and off repeatedly.

Parameters

- **on_time** (*float*) – Number of seconds on. Defaults to 1 second.
- **off_time** (*float*) – Number of seconds off. Defaults to 1 second.
- **fade_in_time** (*float*) – Number of seconds to spend fading in. Defaults to 0.
- **fade_out_time** (*float*) – Number of seconds to spend fading out. Defaults to 0.
- **n** (*int*) – Number of times to blink; `None` (the default) means forever.
- **background** (*bool*) – If `True` (the default), start a background thread to continue blinking and return immediately. If `False`, only return when the blink is finished (warning: the default value of *n* will result in this method never returning).

close()
Shut down the device and release all associated resources. This method can be called on an already closed device without raising an exception.

This method is primarily intended for interactive use at the command line. It disables the device and releases its pin(s) for use by another device.

You can attempt to do this simply by deleting an object, but unless you've cleaned up all references to the object this may not work (even if you've cleaned up all references, there's still no guarantee the garbage collector will actually delete the object at that point). By contrast, the close method provides a means of ensuring that the object is shut down.

For example, if you have a breadboard with a buzzer connected to pin 16, but then wish to attach an LED instead:

```
>>> from gpiozero import *
>>> bz = Buzzer(16)
>>> bz.on()
>>> bz.off()
>>> bz.close()
>>> led = LED(16)
>>> led.blink()
```

Device descendents can also be used as context managers using the `with` statement. For example:

```
>>> from gpiozero import *
>>> with Buzzer(16) as bz:
...     bz.on()
...
>>> with LED(16) as led:
...     led.on()
...
```

off()
Turns the device off.

on()
Turns the device on.

pulse(*fade_in_time=1*, *fade_out_time=1*, *n=None*, *background=True*)
Make the device fade in and out repeatedly.

Parameters

- **fade_in_time** (*float*) – Number of seconds to spend fading in. Defaults to 1.
- **fade_out_time** (*float*) – Number of seconds to spend fading out. Defaults to 1.
- **n** (*int*) – Number of times to pulse; `None` (the default) means forever.
- **background** (*bool*) – If `True` (the default), start a background thread to continue pulsing and return immediately. If `False`, only return when the pulse is finished (warning: the default value of *n* will result in this method never returning).

toggle()
Toggle the state of the device. If the device is currently off (`value` is 0.0), this changes it to "fully" on (`value` is 1.0). If the device has a duty cycle (`value`) of 0.1, this will toggle it to 0.9, and so on.

frequency
The frequency of the pulses used with the PWM device, in Hz. The default is 100Hz.

is_active
Returns `True` if the device is currently active (`value` is non-zero) and `False` otherwise.

value
The duty cycle of the PWM device. 0.0 is off, 1.0 is fully on. Values in between may be specified for varying levels of power in the device.

OutputDevice

class gpiozero.**OutputDevice**(*pin*, *active_high=True*, *initial_value=False*)
Represents a generic GPIO output device.

This class extends `GPIODevice` to add facilities common to GPIO output devices: an `on()` method to switch the device on, a corresponding `off()` method, and a `toggle()` method.

Parameters

- **pin** (*int*) – The GPIO pin (in BCM numbering) that the device is connected to. If this is `None` a `GPIOPinMissing` will be raised.
- **active_high** (*bool*) – If `True` (the default), the `on()` method will set the GPIO to HIGH. If `False`, the `on()` method will set the GPIO to LOW (the `off()` method always does the opposite).
- **initial_value** (*bool*) – If `False` (the default), the device will be off initially. If `None`, the device will be left in whatever state the pin is found in when configured for output (warning: this can be on). If `True`, the device will be switched on initially.

off()
Turns the device off.

on()
Turns the device on.

toggle()
Reverse the state of the device. If it's on, turn it off; if it's off, turn it on.

active_high
When True, the *value* property is True when the device's pin is high. When False the *value* property is True when the device's pin is low (i.e. the value is inverted).

This property can be set after construction; be warned that changing it will invert *value* (i.e. changing this property doesn't change the device's pin state - it just changes how that state is interpreted).

value
Returns True if the device is currently active and False otherwise. Setting this property changes the state of the device.

GPIODevice

class gpiozero.**GPIODevice**(*pin*)
Extends *Device*. Represents a generic GPIO device and provides the services common to all single-pin GPIO devices (like ensuring two GPIO devices do no share a *pin*).

Parameters pin (*int*) – The GPIO pin (in BCM numbering) that the device is connected to. If this is None, *GPIOPinMissing* will be raised. If the pin is already in use by another device, *GPIOPinInUse* will be raised.

close()
Shut down the device and release all associated resources. This method can be called on an already closed device without raising an exception.

This method is primarily intended for interactive use at the command line. It disables the device and releases its pin(s) for use by another device.

You can attempt to do this simply by deleting an object, but unless you've cleaned up all references to the object this may not work (even if you've cleaned up all references, there's still no guarantee the garbage collector will actually delete the object at that point). By contrast, the close method provides a means of ensuring that the object is shut down.

For example, if you have a breadboard with a buzzer connected to pin 16, but then wish to attach an LED instead:

```
>>> from gpiozero import *
>>> bz = Buzzer(16)
>>> bz.on()
>>> bz.off()
>>> bz.close()
>>> led = LED(16)
>>> led.blink()
```

Device descendents can also be used as context managers using the with statement. For example:

```
>>> from gpiozero import *
>>> with Buzzer(16) as bz:
...     bz.on()
...
>>> with LED(16) as led:
...     led.on()
...
```

pin
The *Pin* that the device is connected to. This will be None if the device has been closed (see the

`close()` method). When dealing with GPIO pins, query `pin.number` to discover the GPIO pin (in BCM numbering) that the device is connected to.

SPI Devices

SPI stands for Serial Peripheral Interface and is a mechanism allowing compatible devices to communicate with the Pi. SPI is a four-wire protocol meaning it usually requires four pins to operate:

- A "clock" pin which provides timing information.
- A "MOSI" pin (Master Out, Slave In) which the Pi uses to send information to the device.
- A "MISO" pin (Master In, Slave Out) which the Pi uses to receive information from the device.
- A "select" pin which the Pi uses to indicate which device it's talking to. This last pin is necessary because multiple devices can share the clock, MOSI, and MISO pins, but only one device can be connected to each select pin.

The gpiozero library provides two SPI implementations:

- A software based implementation. This is always available, can use any four GPIO pins for SPI communication, but is rather slow and won't work with all devices.
- A hardware based implementation. This is only available when the SPI kernel module is loaded, and the Python spidev library is available. It can only use specific pins for SPI communication (GPIO11=clock, GPIO10=MOSI, GPIO9=MISO, while GPIO8 is select for device 0 and GPIO7 is select for device 1). However, it is extremely fast and works with all devices.

SPI keyword args

When constructing an SPI device there are two schemes for specifying which pins it is connected to:

- You can specify *port* and *device* keyword arguments. The *port* parameter must be 0 (there is only one user-accessible hardware SPI interface on the Pi using GPIO11 as the clock pin, GPIO10 as the MOSI pin, and GPIO9 as the MISO pin), while the *device* parameter must be 0 or 1. If *device* is 0, the select pin will be GPIO8. If *device* is 1, the select pin will be GPIO7.
- Alternatively you can specify *clock_pin*, *mosi_pin*, *miso_pin*, and *select_pin* keyword arguments. In this case the pins can be any 4 GPIO pins (remember that SPI devices can share clock, MOSI, and MISO pins, but not select pins - the gpiozero library will enforce this restriction).

You cannot mix these two schemes, i.e. attempting to specify *port* and *clock_pin* will result in `SPIBadArgs` being raised. However, you can omit any arguments from either scheme. The defaults are:

- *port* and *device* both default to 0.
- *clock_pin* defaults to 11, *mosi_pin* defaults to 10, *miso_pin* defaults to 9, and *select_pin* defaults to 8.

Hence the following constructors are all equiavlent:

```
from gpiozero import MCP3008

MCP3008(channel=0)
MCP3008(channel=0, device=0)
MCP3008(channel=0, port=0, device=0)
MCP3008(channel=0, select_pin=8)
MCP3008(channel=0, clock_pin=11, mosi_pin=10, miso_pin=9, select_pin=8)
```

Note that the defaults describe equivalent sets of pins and that these pins are compatible with the hardware implementation. Regardless of which scheme you use, gpiozero will attempt to use the hardware implementation if it is available and if the selected pins are compatible, falling back to the software implementation if not.

Analog to Digital Converters (ADC)

class gpiozero.**MCP3001**(***spi_args*)
The MCP3001 is a 10-bit analog to digital converter with 1 channel

value
The current value read from the device, scaled to a value between 0 and 1 (or -1 to +1 for devices operating in differential mode).

class gpiozero.**MCP3002**(*channel=0*, *differential=False*, ***spi_args*)
The MCP3002 is a 10-bit analog to digital converter with 2 channels (0-1).

channel
The channel to read data from. The MCP3008/3208/3304 have 8 channels (0-7), while the MCP3004/3204/3302 have 4 channels (0-3), and the MCP3301 only has 1 channel.

differential
If `True`, the device is operated in pseudo-differential mode. In this mode one channel (specified by the channel attribute) is read relative to the value of a second channel (implied by the chip's design).

Please refer to the device data-sheet to determine which channel is used as the relative base value (for example, when using an *MCP3008* in differential mode, channel 0 is read relative to channel 1).

value
The current value read from the device, scaled to a value between 0 and 1 (or -1 to +1 for devices operating in differential mode).

class gpiozero.**MCP3004**(*channel=0*, *differential=False*, ***spi_args*)
The MCP3004 is a 10-bit analog to digital converter with 4 channels (0-3).

channel
The channel to read data from. The MCP3008/3208/3304 have 8 channels (0-7), while the MCP3004/3204/3302 have 4 channels (0-3), and the MCP3301 only has 1 channel.

differential
If `True`, the device is operated in pseudo-differential mode. In this mode one channel (specified by the channel attribute) is read relative to the value of a second channel (implied by the chip's design).

Please refer to the device data-sheet to determine which channel is used as the relative base value (for example, when using an *MCP3008* in differential mode, channel 0 is read relative to channel 1).

value
The current value read from the device, scaled to a value between 0 and 1 (or -1 to +1 for devices operating in differential mode).

class gpiozero.**MCP3008**(*channel=0*, *differential=False*, ***spi_args*)
The MCP3008 is a 10-bit analog to digital converter with 8 channels (0-7).

channel
The channel to read data from. The MCP3008/3208/3304 have 8 channels (0-7), while the MCP3004/3204/3302 have 4 channels (0-3), and the MCP3301 only has 1 channel.

differential
If `True`, the device is operated in pseudo-differential mode. In this mode one channel (specified by the channel attribute) is read relative to the value of a second channel (implied by the chip's design).

Please refer to the device data-sheet to determine which channel is used as the relative base value (for example, when using an *MCP3008* in differential mode, channel 0 is read relative to channel 1).

value
The current value read from the device, scaled to a value between 0 and 1 (or -1 to +1 for devices operating in differential mode).

class gpiozero.**MCP3201**(***spi_args*)
The MCP3201 is a 12-bit analog to digital converter with 1 channel

value
The current value read from the device, scaled to a value between 0 and 1 (or -1 to +1 for devices operating in differential mode).

class `gpiozero.`**`MCP3202`**(*channel=0*, *differential=False*, ***spi_args*)
The MCP3202 is a 12-bit analog to digital converter with 2 channels (0-1).

channel
The channel to read data from. The MCP3008/3208/3304 have 8 channels (0-7), while the MCP3004/3204/3302 have 4 channels (0-3), and the MCP3301 only has 1 channel.

differential
If `True`, the device is operated in pseudo-differential mode. In this mode one channel (specified by the channel attribute) is read relative to the value of a second channel (implied by the chip's design).

Please refer to the device data-sheet to determine which channel is used as the relative base value (for example, when using an *MCP3008* in differential mode, channel 0 is read relative to channel 1).

value
The current value read from the device, scaled to a value between 0 and 1 (or -1 to +1 for devices operating in differential mode).

class `gpiozero.`**`MCP3204`**(*channel=0*, *differential=False*, ***spi_args*)
The MCP3204 is a 12-bit analog to digital converter with 4 channels (0-3).

channel
The channel to read data from. The MCP3008/3208/3304 have 8 channels (0-7), while the MCP3004/3204/3302 have 4 channels (0-3), and the MCP3301 only has 1 channel.

differential
If `True`, the device is operated in pseudo-differential mode. In this mode one channel (specified by the channel attribute) is read relative to the value of a second channel (implied by the chip's design).

Please refer to the device data-sheet to determine which channel is used as the relative base value (for example, when using an *MCP3008* in differential mode, channel 0 is read relative to channel 1).

value
The current value read from the device, scaled to a value between 0 and 1 (or -1 to +1 for devices operating in differential mode).

class `gpiozero.`**`MCP3208`**(*channel=0*, *differential=False*, ***spi_args*)
The MCP3208 is a 12-bit analog to digital converter with 8 channels (0-7).

channel
The channel to read data from. The MCP3008/3208/3304 have 8 channels (0-7), while the MCP3004/3204/3302 have 4 channels (0-3), and the MCP3301 only has 1 channel.

differential
If `True`, the device is operated in pseudo-differential mode. In this mode one channel (specified by the channel attribute) is read relative to the value of a second channel (implied by the chip's design).

Please refer to the device data-sheet to determine which channel is used as the relative base value (for example, when using an *MCP3008* in differential mode, channel 0 is read relative to channel 1).

value
The current value read from the device, scaled to a value between 0 and 1 (or -1 to +1 for devices operating in differential mode).

class `gpiozero.`**`MCP3301`**(***spi_args*)
The MCP3301 is a signed 13-bit analog to digital converter. Please note that the MCP3301 always operates in differential mode between its two channels and the output value is scaled from -1 to +1.

value
The current value read from the device, scaled to a value between 0 and 1 (or -1 to +1 for devices operating in differential mode).

class `gpiozero.`**`MCP3302`**(*channel=0*, *differential=False*, ***spi_args*)

The MCP3302 is a 12/13-bit analog to digital converter with 4 channels (0-3). When operated in differential mode, the device outputs a signed 13-bit value which is scaled from -1 to +1. When operated in single-ended mode (the default), the device outputs an unsigned 12-bit value scaled from 0 to 1.

`channel`

The channel to read data from. The MCP3008/3208/3304 have 8 channels (0-7), while the MCP3004/3204/3302 have 4 channels (0-3), and the MCP3301 only has 1 channel.

`differential`

If `True`, the device is operated in pseudo-differential mode. In this mode one channel (specified by the channel attribute) is read relative to the value of a second channel (implied by the chip's design).

Please refer to the device data-sheet to determine which channel is used as the relative base value (for example, when using an *`MCP3008`* in differential mode, channel 0 is read relative to channel 1).

`value`

The current value read from the device, scaled to a value between 0 and 1 (or -1 to +1 for devices operating in differential mode).

class `gpiozero.`**`MCP3304`**(*channel=0*, *differential=False*, ***spi_args*)

The MCP3304 is a 12/13-bit analog to digital converter with 8 channels (0-7). When operated in differential mode, the device outputs a signed 13-bit value which is scaled from -1 to +1. When operated in single-ended mode (the default), the device outputs an unsigned 12-bit value scaled from 0 to 1.

`channel`

The channel to read data from. The MCP3008/3208/3304 have 8 channels (0-7), while the MCP3004/3204/3302 have 4 channels (0-3), and the MCP3301 only has 1 channel.

`differential`

If `True`, the device is operated in pseudo-differential mode. In this mode one channel (specified by the channel attribute) is read relative to the value of a second channel (implied by the chip's design).

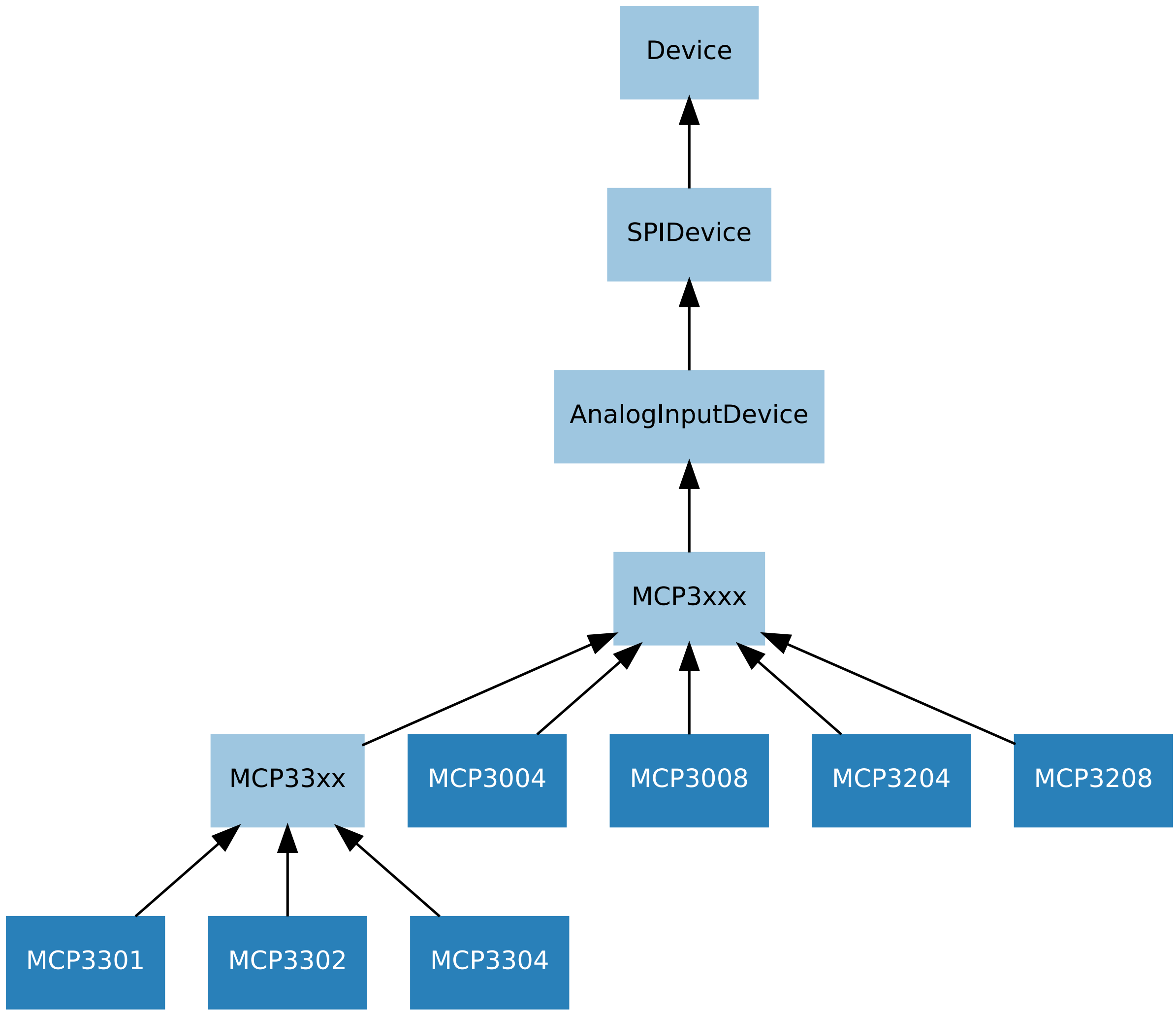
Please refer to the device data-sheet to determine which channel is used as the relative base value (for example, when using an *`MCP3008`* in differential mode, channel 0 is read relative to channel 1).

`value`

The current value read from the device, scaled to a value between 0 and 1 (or -1 to +1 for devices operating in differential mode).

Base Classes

The classes in the sections above are derived from a series of base classes, some of which are effectively abstract. The classes form the (partial) hierarchy displayed in the graph below:



The following sections document these base classes for advanced users that wish to construct classes for their own devices.

AnalogInputDevice

class gpiozero.**AnalogInputDevice**(*bits=None*, ***spi_args*)

Represents an analog input device connected to SPI (serial interface).

Typical analog input devices are analog to digital converters (ADCs). Several classes are provided for specific ADC chips, including *MCP3004*, *MCP3008*, *MCP3204*, and *MCP3208*.

The following code demonstrates reading the first channel of an MCP3008 chip attached to the Pi's SPI pins:

```
from gpiozero import MCP3008

pot = MCP3008(0)
print(pot.value)
```

The *value* attribute is normalized such that its value is always between 0.0 and 1.0 (or in special cases, such as differential sampling, -1 to +1). Hence, you can use an analog input to control the brightness of a *PWMLED* like so:

```
from gpiozero import MCP3008, PWMLED

pot = MCP3008(0)
```

```
led = PWMLED(17)
led.source = pot.values
```

bits
The bit-resolution of the device/channel.

raw_value
The raw value as read from the device.

value
The current value read from the device, scaled to a value between 0 and 1 (or -1 to +1 for devices operating in differential mode).

SPIDevice

class gpiozero.**SPIDevice**(***spi_args*)
Extends `Device`. Represents a device that communicates via the SPI protocol.

See *SPI keyword args* for information on the keyword arguments that can be specified with the constructor.

close()
Shut down the device and release all associated resources. This method can be called on an already closed device without raising an exception.

This method is primarily intended for interactive use at the command line. It disables the device and releases its pin(s) for use by another device.

You can attempt to do this simply by deleting an object, but unless you've cleaned up all references to the object this may not work (even if you've cleaned up all references, there's still no guarantee the garbage collector will actually delete the object at that point). By contrast, the close method provides a means of ensuring that the object is shut down.

For example, if you have a breadboard with a buzzer connected to pin 16, but then wish to attach an LED instead:

```
>>> from gpiozero import *
>>> bz = Buzzer(16)
>>> bz.on()
>>> bz.off()
>>> bz.close()
>>> led = LED(16)
>>> led.blink()
```

`Device` descendents can also be used as context managers using the `with` statement. For example:

```
>>> from gpiozero import *
>>> with Buzzer(16) as bz:
...     bz.on()
...
>>> with LED(16) as led:
...     led.on()
...
```

Boards and Accessories

These additional interfaces are provided to group collections of components together for ease of use, and as examples. They are composites made up of components from the various *Input Devices* and *Output Devices* provided by GPIO Zero. See those pages for more information on using components individually.

Note: All GPIO pin numbers use Broadcom (BCM) numbering. See the *Recipes* page for more information.

LEDBoard

class `gpiozero.`**`LEDBoard`**(**pins, pwm=False, active_high=True, initial_value=False, **named_pins*)

Extends *`LEDCollection`* and represents a generic LED board or collection of LEDs.

The following example turns on all the LEDs on a board containing 5 LEDs attached to GPIO pins 2 through 6:

```
from gpiozero import LEDBoard

leds = LEDBoard(2, 3, 4, 5, 6)
leds.on()
```

Parameters

- **`*pins`** (*`int`*) – Specify the GPIO pins that the LEDs of the board are attached to. You can designate as many pins as necessary. You can also specify *`LEDBoard`* instances to create trees of LEDs.
- **`pwm`** (*`bool`*) – If `True`, construct *`PWMLED`* instances for each pin. If `False` (the default), construct regular *`LED`* instances. This parameter can only be specified as a keyword parameter.
- **`active_high`** (*`bool`*) – If `True` (the default), the *`on()`* method will set all the associated pins to HIGH. If `False`, the *`on()`* method will set all pins to LOW (the *`off()`* method always does the opposite). This parameter can only be specified as a keyword parameter.
- **`initial_value`** (*`bool`*) – If `False` (the default), all LEDs will be off initially. If `None`, each device will be left in whatever state the pin is found in when configured for output (warning: this can be on). If `True`, the device will be switched on initially. This parameter can only be specified as a keyword parameter.
- **`**named_pins`** – Specify GPIO pins that LEDs of the board are attached to, associating each LED with a property name. You can designate as many pins as necessary and use any names, provided they're not already in use by something else. You can also specify *`LEDBoard`* instances to create trees of LEDs.

`blink`(*on_time=1, off_time=1, fade_in_time=0, fade_out_time=0, n=None, background=True*)

Make all the LEDs turn on and off repeatedly.

Parameters

- **`on_time`** (*`float`*) – Number of seconds on. Defaults to 1 second.
- **`off_time`** (*`float`*) – Number of seconds off. Defaults to 1 second.
- **`fade_in_time`** (*`float`*) – Number of seconds to spend fading in. Defaults to 0. Must be 0 if `pwm` was `False` when the class was constructed (`ValueError` will be raised if not).
- **`fade_out_time`** (*`float`*) – Number of seconds to spend fading out. Defaults to 0. Must be 0 if `pwm` was `False` when the class was constructed (`ValueError` will be raised if not).
- **`n`** (*`int`*) – Number of times to blink; `None` (the default) means forever.

- **background** (*bool*) – If `True`, start a background thread to continue blinking and return immediately. If `False`, only return when the blink is finished (warning: the default value of *n* will result in this method never returning).

close()
Shut down the device and release all associated resources. This method can be called on an already closed device without raising an exception.

This method is primarily intended for interactive use at the command line. It disables the device and releases its pin(s) for use by another device.

You can attempt to do this simply by deleting an object, but unless you've cleaned up all references to the object this may not work (even if you've cleaned up all references, there's still no guarantee the garbage collector will actually delete the object at that point). By contrast, the close method provides a means of ensuring that the object is shut down.

For example, if you have a breadboard with a buzzer connected to pin 16, but then wish to attach an LED instead:

```
>>> from gpiozero import *
>>> bz = Buzzer(16)
>>> bz.on()
>>> bz.off()
>>> bz.close()
>>> led = LED(16)
>>> led.blink()
```

`Device` descendents can also be used as context managers using the `with` statement. For example:

```
>>> from gpiozero import *
>>> with Buzzer(16) as bz:
...     bz.on()
...
>>> with LED(16) as led:
...     led.on()
...
```

off(**args*)
Turn all the output devices off.

on(**args*)
Turn all the output devices on.

pulse(*fade_in_time=1*, *fade_out_time=1*, *n=None*, *background=True*)
Make the device fade in and out repeatedly.

Parameters

- **fade_in_time** (*float*) – Number of seconds to spend fading in. Defaults to 1.
- **fade_out_time** (*float*) – Number of seconds to spend fading out. Defaults to 1.
- **n** (*int*) – Number of times to blink; `None` (the default) means forever.
- **background** (*bool*) – If `True` (the default), start a background thread to continue blinking and return immediately. If `False`, only return when the blink is finished (warning: the default value of *n* will result in this method never returning).

toggle(**args*)
Toggle all the output devices. For each device, if it's on, turn it off; if it's off, turn it on.

leds
A flat tuple of all LEDs contained in this collection (and all sub-collections).

source
The iterable to use as a source of values for `value`.

source_delay
The delay (measured in seconds) in the loop used to read values from *source*. Defaults to 0.01 seconds which is generally sufficient to keep CPU usage to a minimum while providing adequate responsiveness.

value
A tuple containing a value for each subordinate device. This property can also be set to update the state of all subordinate output devices.

values
An infinite iterator of values read from *value*.

LEDBarGraph

class gpiozero.**LEDBarGraph**(**pins*, *initial_value=0*)
Extends *LEDCollection* to control a line of LEDs representing a bar graph. Positive values (0 to 1) light the LEDs from first to last. Negative values (-1 to 0) light the LEDs from last to first.

The following example demonstrates turning on the first two and last two LEDs in a board containing five LEDs attached to GPIOs 2 through 6:

```
from gpiozero import LEDBarGraph
from time import sleep

graph = LEDBarGraph(2, 3, 4, 5, 6)
graph.value = 2/5  # Light the first two LEDs only
sleep(1)
graph.value = -2/5 # Light the last two LEDs only
sleep(1)
graph.off()
```

As with other output devices, *source* and *values* are supported:

```
from gpiozero import LEDBarGraph, MCP3008
from signal import pause

graph = LEDBarGraph(2, 3, 4, 5, 6, pwm=True)
pot = MCP3008(channel=0)
graph.source = pot.values
pause()
```

Parameters

- ***pins** (*int*) – Specify the GPIO pins that the LEDs of the bar graph are attached to. You can designate as many pins as necessary.
- **pwm** (*bool*) – If True, construct *PWMLED* instances for each pin. If False (the default), construct regular *LED* instances. This parameter can only be specified as a keyword parameter.
- **active_high** (*bool*) – If True (the default), the *on()* method will set all the associated pins to HIGH. If False, the *on()* method will set all pins to LOW (the *off()* method always does the opposite). This parameter can only be specified as a keyword parameter.
- **initial_value** (*float*) – The initial *value* of the graph given as a float between -1 and +1. Defaults to 0.0. This parameter can only be specified as a keyword parameter.

off()
Turn all the output devices off.

on()
Turn all the output devices on.

toggle()
Toggle all the output devices. For each device, if it's on, turn it off; if it's off, turn it on.

leds
A flat tuple of all LEDs contained in this collection (and all sub-collections).

source
The iterable to use as a source of values for `value`.

source_delay
The delay (measured in seconds) in the loop used to read values from `source`. Defaults to 0.01 seconds which is generally sufficient to keep CPU usage to a minimum while providing adequate responsiveness.

value
The value of the LED bar graph. When no LEDs are lit, the value is 0. When all LEDs are lit, the value is 1. Values between 0 and 1 light LEDs linearly from first to last. Values between 0 and -1 light LEDs linearly from last to first.

To light a particular number of LEDs, simply divide that number by the number of LEDs. For example, if your graph contains 3 LEDs, the following will light the first:

```
from gpiozero import LEDBarGraph

graph = LEDBarGraph(12, 16, 19)
graph.value = 1/3
```

Note: Setting value to -1 will light all LEDs. However, querying it subsequently will return 1 as both representations are the same in hardware. The readable range of `value` is effectively -1 < value <= 1.

values
An infinite iterator of values read from *value*.

ButtonBoard

class gpiozero.**ButtonBoard**(**pins, pull_up=True, bounce_time=None, hold_time=1, hold_repeat=False, **named_pins*)
Extends `CompositeDevice` and represents a generic button board or collection of buttons.

Parameters

- ***pins** (*int*) – Specify the GPIO pins that the buttons of the board are attached to. You can designate as many pins as necessary.
- **pull_up** (*bool*) – If `True` (the default), the GPIO pins will be pulled high by default. In this case, connect the other side of the buttons to ground. If `False`, the GPIO pins will be pulled low by default. In this case, connect the other side of the buttons to 3V3. This parameter can only be specified as a keyword parameter.
- **bounce_time** (*float*) – If `None` (the default), no software bounce compensation will be performed. Otherwise, this is the length of time (in seconds) that the buttons will ignore changes in state after an initial change. This parameter can only be specified as a keyword parameter.
- **hold_time** (*float*) – The length of time (in seconds) to wait after any button is pushed, until executing the `when_held` handler. Defaults to `1`. This parameter can only be specified as a keyword parameter.

- **hold_repeat** (*bool*) – If True, the *when_held* handler will be repeatedly executed as long as any buttons remain held, every *hold_time* seconds. If False (the default) the *when_held* handler will be only be executed once per hold. This parameter can only be specified as a keyword parameter.
- ****named_pins** – Specify GPIO pins that buttons of the board are attached to, associating each button with a property name. You can designate as many pins as necessary and use any names, provided they're not already in use by something else.

wait_for_active (*timeout=None*)
Pause the script until the device is activated, or the timeout is reached.

Parameters **timeout** (*float*) – Number of seconds to wait before proceeding. If this is None (the default), then wait indefinitely until the device is active.

wait_for_inactive (*timeout=None*)
Pause the script until the device is deactivated, or the timeout is reached.

Parameters **timeout** (*float*) – Number of seconds to wait before proceeding. If this is None (the default), then wait indefinitely until the device is inactive.

wait_for_press (*timeout=None*)
Pause the script until the device is activated, or the timeout is reached.

Parameters **timeout** (*float*) – Number of seconds to wait before proceeding. If this is None (the default), then wait indefinitely until the device is active.

wait_for_release (*timeout=None*)
Pause the script until the device is deactivated, or the timeout is reached.

Parameters **timeout** (*float*) – Number of seconds to wait before proceeding. If this is None (the default), then wait indefinitely until the device is inactive.

active_time
The length of time (in seconds) that the device has been active for. When the device is inactive, this is None.

held_time
The length of time (in seconds) that the device has been held for. This is counted from the first execution of the *when_held* event rather than when the device activated, in contrast to *active_time*. If the device is not currently held, this is None.

hold_repeat
If True, *when_held* will be executed repeatedly with *hold_time* seconds between each invocation.

hold_time
The length of time (in seconds) to wait after the device is activated, until executing the *when_held* handler. If *hold_repeat* is True, this is also the length of time between invocations of *when_held*.

inactive_time
The length of time (in seconds) that the device has been inactive for. When the device is active, this is None.

is_held
When True, the device has been active for at least *hold_time* seconds.

pressed_time
The length of time (in seconds) that the device has been active for. When the device is inactive, this is None.

pull_up
If True, the device uses a pull-up resistor to set the GPIO pin "high" by default.

values
An infinite iterator of values read from *value*.

when_activated
The function to run when the device changes state from inactive to active.

This can be set to a function which accepts no (mandatory) parameters, or a Python function which accepts a single mandatory parameter (with as many optional parameters as you like). If the function accepts a single mandatory parameter, the device that activated will be passed as that parameter.

Set this property to `None` (the default) to disable the event.

when_deactivated
The function to run when the device changes state from active to inactive.

This can be set to a function which accepts no (mandatory) parameters, or a Python function which accepts a single mandatory parameter (with as many optional parameters as you like). If the function accepts a single mandatory parameter, the device that deactivated will be passed as that parameter.

Set this property to `None` (the default) to disable the event.

when_held
The function to run when the device has remained active for *hold_time* seconds.

This can be set to a function which accepts no (mandatory) parameters, or a Python function which accepts a single mandatory parameter (with as many optional parameters as you like). If the function accepts a single mandatory parameter, the device that activated will be passed as that parameter.

Set this property to `None` (the default) to disable the event.

when_pressed
The function to run when the device changes state from inactive to active.

This can be set to a function which accepts no (mandatory) parameters, or a Python function which accepts a single mandatory parameter (with as many optional parameters as you like). If the function accepts a single mandatory parameter, the device that activated will be passed as that parameter.

Set this property to `None` (the default) to disable the event.

when_released
The function to run when the device changes state from active to inactive.

This can be set to a function which accepts no (mandatory) parameters, or a Python function which accepts a single mandatory parameter (with as many optional parameters as you like). If the function accepts a single mandatory parameter, the device that deactivated will be passed as that parameter.

Set this property to `None` (the default) to disable the event.

TrafficLights

class gpiozero.**TrafficLights**(*red=None, amber=None, green=None, pwm=False, initial_value=False, yellow=None*)
Extends *LEDBoard* for devices containing red, yellow, and green LEDs.

The following example initializes a device connected to GPIO pins 2, 3, and 4, then lights the amber (yellow) LED attached to GPIO 3:

```
from gpiozero import TrafficLights

traffic = TrafficLights(2, 3, 4)
traffic.amber.on()
```

Parameters

- **red** (*int*) – The GPIO pin that the red LED is attached to.
- **amber** (*int*) – The GPIO pin that the amber LED is attached to.
- **green** (*int*) – The GPIO pin that the green LED is attached to.

- **pwm** (*bool*) – If True, construct *PWMLED* instances to represent each LED. If False (the default), construct regular *LED* instances.
- **initial_value** (*bool*) – If False (the default), all LEDs will be off initially. If None, each device will be left in whatever state the pin is found in when configured for output (warning: this can be on). If True, the device will be switched on initially.
- **yellow** (*int*) – The GPIO pin that the yellow LED is attached to. This is merely an alias for the amber parameter - you can't specify both amber and yellow.

blink (*on_time=1, off_time=1, fade_in_time=0, fade_out_time=0, n=None, background=True*)

Make all the LEDs turn on and off repeatedly.

Parameters

- **on_time** (*float*) – Number of seconds on. Defaults to 1 second.
- **off_time** (*float*) – Number of seconds off. Defaults to 1 second.
- **fade_in_time** (*float*) – Number of seconds to spend fading in. Defaults to 0. Must be 0 if pwm was False when the class was constructed (ValueError will be raised if not).
- **fade_out_time** (*float*) – Number of seconds to spend fading out. Defaults to 0. Must be 0 if pwm was False when the class was constructed (ValueError will be raised if not).
- **n** (*int*) – Number of times to blink; None (the default) means forever.
- **background** (*bool*) – If True, start a background thread to continue blinking and return immediately. If False, only return when the blink is finished (warning: the default value of *n* will result in this method never returning).

close ()

Shut down the device and release all associated resources. This method can be called on an already closed device without raising an exception.

This method is primarily intended for interactive use at the command line. It disables the device and releases its pin(s) for use by another device.

You can attempt to do this simply by deleting an object, but unless you've cleaned up all references to the object this may not work (even if you've cleaned up all references, there's still no guarantee the garbage collector will actually delete the object at that point). By contrast, the close method provides a means of ensuring that the object is shut down.

For example, if you have a breadboard with a buzzer connected to pin 16, but then wish to attach an LED instead:

```
>>> from gpiozero import *
>>> bz = Buzzer(16)
>>> bz.on()
>>> bz.off()
>>> bz.close()
>>> led = LED(16)
>>> led.blink()
```

Device descendents can also be used as context managers using the with statement. For example:

```
>>> from gpiozero import *
>>> with Buzzer(16) as bz:
...     bz.on()
...
>>> with LED(16) as led:
...     led.on()
...
```

off (**args*)
Turn all the output devices off.

on (**args*)
Turn all the output devices on.

pulse (*fade_in_time=1, fade_out_time=1, n=None, background=True*)
Make the device fade in and out repeatedly.

Parameters

- **fade_in_time** (*float*) – Number of seconds to spend fading in. Defaults to 1.
- **fade_out_time** (*float*) – Number of seconds to spend fading out. Defaults to 1.
- **n** (*int*) – Number of times to blink; `None` (the default) means forever.
- **background** (*bool*) – If `True` (the default), start a background thread to continue blinking and return immediately. If `False`, only return when the blink is finished (warning: the default value of *n* will result in this method never returning).

toggle (**args*)
Toggle all the output devices. For each device, if it's on, turn it off; if it's off, turn it on.

leds
A flat tuple of all LEDs contained in this collection (and all sub-collections).

source
The iterable to use as a source of values for `value`.

source_delay
The delay (measured in seconds) in the loop used to read values from `source`. Defaults to 0.01 seconds which is generally sufficient to keep CPU usage to a minimum while providing adequate responsiveness.

value
A tuple containing a value for each subordinate device. This property can also be set to update the state of all subordinate output devices.

values
An infinite iterator of values read from *value*.

LedBorg

class gpiozero.**LedBorg** (*initial_value=(0, 0, 0), pwm=True*)
Extends `RGBLED` for the PiBorg LedBorg: an add-on board containing a very bright RGB LED.

The LedBorg pins are fixed and therefore there's no need to specify them when constructing this class. The following example turns the LedBorg purple:

```
from gpiozero import LedBorg

led = LedBorg()
led.color = (1, 0, 1)
```

Parameters

- **initial_value** (*tuple*) – The initial color for the LedBorg. Defaults to black `(0, 0, 0)`.
- **pwm** (*bool*) – If `True` (the default), construct `PWMLED` instances for each component of the LedBorg. If `False`, construct regular `LED` instances, which prevents smooth color graduations.

blink(*on_time=1*, *off_time=1*, *fade_in_time=0*, *fade_out_time=0*, *on_color=(1, 1, 1)*, *off_color=(0, 0, 0)*, *n=None*, *background=True*)

Make the device turn on and off repeatedly.

Parameters

- **on_time** (*float*) – Number of seconds on. Defaults to 1 second.
- **off_time** (*float*) – Number of seconds off. Defaults to 1 second.
- **fade_in_time** (*float*) – Number of seconds to spend fading in. Defaults to 0. Must be 0 if `pwm` was `False` when the class was constructed (`ValueError` will be raised if not).
- **fade_out_time** (*float*) – Number of seconds to spend fading out. Defaults to 0. Must be 0 if `pwm` was `False` when the class was constructed (`ValueError` will be raised if not).
- **on_color** (*tuple*) – The color to use when the LED is "on". Defaults to white.
- **off_color** (*tuple*) – The color to use when the LED is "off". Defaults to black.
- **n** (*int*) – Number of times to blink; `None` (the default) means forever.
- **background** (*bool*) – If `True` (the default), start a background thread to continue blinking and return immediately. If `False`, only return when the blink is finished (warning: the default value of *n* will result in this method never returning).

close()

Shut down the device and release all associated resources. This method can be called on an already closed device without raising an exception.

This method is primarily intended for interactive use at the command line. It disables the device and releases its pin(s) for use by another device.

You can attempt to do this simply by deleting an object, but unless you've cleaned up all references to the object this may not work (even if you've cleaned up all references, there's still no guarantee the garbage collector will actually delete the object at that point). By contrast, the close method provides a means of ensuring that the object is shut down.

For example, if you have a breadboard with a buzzer connected to pin 16, but then wish to attach an LED instead:

```
>>> from gpiozero import *
>>> bz = Buzzer(16)
>>> bz.on()
>>> bz.off()
>>> bz.close()
>>> led = LED(16)
>>> led.blink()
```

`Device` descendents can also be used as context managers using the `with` statement. For example:

```
>>> from gpiozero import *
>>> with Buzzer(16) as bz:
...     bz.on()
...
>>> with LED(16) as led:
...     led.on()
...
```

off()

Turn the LED off. This is equivalent to setting the LED color to black `(0, 0, 0)`.

on()

Turn the LED on. This equivalent to setting the LED color to white `(1, 1, 1)`.

pulse (*fade_in_time=1*, *fade_out_time=1*, *on_color=(1, 1, 1)*, *off_color=(0, 0, 0)*, *n=None*, *background=True*)
Make the device fade in and out repeatedly.

Parameters

- **fade_in_time** (*float*) – Number of seconds to spend fading in. Defaults to 1.
- **fade_out_time** (*float*) – Number of seconds to spend fading out. Defaults to 1.
- **on_color** (*tuple*) – The color to use when the LED is "on". Defaults to white.
- **off_color** (*tuple*) – The color to use when the LED is "off". Defaults to black.
- **n** (*int*) – Number of times to pulse; `None` (the default) means forever.
- **background** (*bool*) – If `True` (the default), start a background thread to continue pulsing and return immediately. If `False`, only return when the pulse is finished (warning: the default value of *n* will result in this method never returning).

toggle ()
Toggle the state of the device. If the device is currently off (`value` is `(0, 0, 0)`), this changes it to "fully" on (`value` is `(1, 1, 1)`). If the device has a specific color, this method inverts the color.

color
Represents the color of the LED as an RGB 3-tuple of `(red, green, blue)` where each value is between 0 and 1 if `pwm` was `True` when the class was constructed (and only 0 or 1 if not).

For example, purple would be `(1, 0, 1)` and yellow would be `(1, 1, 0)`, while orange would be `(1, 0.5, 0)`.

is_active
Returns `True` if the LED is currently active (not black) and `False` otherwise.

is_lit
Returns `True` if the LED is currently active (not black) and `False` otherwise.

source
The iterable to use as a source of values for `value`.

source_delay
The delay (measured in seconds) in the loop used to read values from `source`. Defaults to 0.01 seconds which is generally sufficient to keep CPU usage to a minimum while providing adequate responsiveness.

value
Represents the color of the LED as an RGB 3-tuple of `(red, green, blue)` where each value is between 0 and 1 if `pwm` was `True` when the class was constructed (and only 0 or 1 if not).

For example, purple would be `(1, 0, 1)` and yellow would be `(1, 1, 0)`, while orange would be `(1, 0.5, 0)`.

values
An infinite iterator of values read from *value*.

PiLITEr

class gpiozero.**PiLiter** (*pwm=False*, *initial_value=False*)
Extends `LEDBoard` for the Ciseco Pi-LITEr: a strip of 8 very bright LEDs.

The Pi-LITEr pins are fixed and therefore there's no need to specify them when constructing this class. The following example turns on all the LEDs of the Pi-LITEr:

```
from gpiozero import PiLiter

lite = PiLiter()
lite.on()
```

Parameters

- **pwm** (*bool*) – If `True`, construct *PWMLED* instances for each pin. If `False` (the default), construct regular *LED* instances.
- **initial_value** (*bool*) – If `False` (the default), all LEDs will be off initially. If `None`, each device will be left in whatever state the pin is found in when configured for output (warning: this can be on). If `True`, the device will be switched on initially.

blink (*on_time=1*, *off_time=1*, *fade_in_time=0*, *fade_out_time=0*, *n=None*, *background=True*)

Make all the LEDs turn on and off repeatedly.

Parameters

- **on_time** (*float*) – Number of seconds on. Defaults to 1 second.
- **off_time** (*float*) – Number of seconds off. Defaults to 1 second.
- **fade_in_time** (*float*) – Number of seconds to spend fading in. Defaults to 0. Must be 0 if `pwm` was `False` when the class was constructed (`ValueError` will be raised if not).
- **fade_out_time** (*float*) – Number of seconds to spend fading out. Defaults to 0. Must be 0 if `pwm` was `False` when the class was constructed (`ValueError` will be raised if not).
- **n** (*int*) – Number of times to blink; `None` (the default) means forever.
- **background** (*bool*) – If `True`, start a background thread to continue blinking and return immediately. If `False`, only return when the blink is finished (warning: the default value of *n* will result in this method never returning).

close ()

Shut down the device and release all associated resources. This method can be called on an already closed device without raising an exception.

This method is primarily intended for interactive use at the command line. It disables the device and releases its pin(s) for use by another device.

You can attempt to do this simply by deleting an object, but unless you've cleaned up all references to the object this may not work (even if you've cleaned up all references, there's still no guarantee the garbage collector will actually delete the object at that point). By contrast, the close method provides a means of ensuring that the object is shut down.

For example, if you have a breadboard with a buzzer connected to pin 16, but then wish to attach an LED instead:

```
>>> from gpiozero import *
>>> bz = Buzzer(16)
>>> bz.on()
>>> bz.off()
>>> bz.close()
>>> led = LED(16)
>>> led.blink()
```

Device descendents can also be used as context managers using the `with` statement. For example:

```
>>> from gpiozero import *
>>> with Buzzer(16) as bz:
...     bz.on()
...
>>> with LED(16) as led:
...     led.on()
...
```

off (**args*)
Turn all the output devices off.

on (**args*)
Turn all the output devices on.

pulse (*fade_in_time=1*, *fade_out_time=1*, *n=None*, *background=True*)
Make the device fade in and out repeatedly.

Parameters

- **fade_in_time** (*float*) – Number of seconds to spend fading in. Defaults to 1.
- **fade_out_time** (*float*) – Number of seconds to spend fading out. Defaults to 1.
- **n** (*int*) – Number of times to blink; `None` (the default) means forever.
- **background** (*bool*) – If `True` (the default), start a background thread to continue blinking and return immediately. If `False`, only return when the blink is finished (warning: the default value of *n* will result in this method never returning).

toggle (**args*)
Toggle all the output devices. For each device, if it's on, turn it off; if it's off, turn it on.

leds
A flat tuple of all LEDs contained in this collection (and all sub-collections).

source
The iterable to use as a source of values for `value`.

source_delay
The delay (measured in seconds) in the loop used to read values from `source`. Defaults to 0.01 seconds which is generally sufficient to keep CPU usage to a minimum while providing adequate responsiveness.

value
A tuple containing a value for each subordinate device. This property can also be set to update the state of all subordinate output devices.

values
An infinite iterator of values read from *value*.

PiLITEr Bar Graph

class gpiozero.**PiLiterBarGraph** (*pwm=False*, *initial_value=0.0*)
Extends `LEDBarGraph` to treat the Ciseco Pi-LITEr as an 8-segment bar graph.

The Pi-LITEr pins are fixed and therefore there's no need to specify them when constructing this class. The following example sets the graph value to 0.5:

```
from gpiozero import PiLiterBarGraph

graph = PiLiterBarGraph()
graph.value = 0.5
```

Parameters

- **pwm** (*bool*) – If `True`, construct `PWMLED` instances for each pin. If `False` (the default), construct regular `LED` instances.
- **initial_value** (*float*) – The initial `value` of the graph given as a float between -1 and +1. Defaults to `0.0`.

off ()
Turn all the output devices off.

on()
Turn all the output devices on.

toggle()
Toggle all the output devices. For each device, if it's on, turn it off; if it's off, turn it on.

leds
A flat tuple of all LEDs contained in this collection (and all sub-collections).

source
The iterable to use as a source of values for `value`.

source_delay
The delay (measured in seconds) in the loop used to read values from `source`. Defaults to 0.01 seconds which is generally sufficient to keep CPU usage to a minimum while providing adequate responsiveness.

value
The value of the LED bar graph. When no LEDs are lit, the value is 0. When all LEDs are lit, the value is 1. Values between 0 and 1 light LEDs linearly from first to last. Values between 0 and -1 light LEDs linearly from last to first.

To light a particular number of LEDs, simply divide that number by the number of LEDs. For example, if your graph contains 3 LEDs, the following will light the first:

```
from gpiozero import LEDBarGraph

graph = LEDBarGraph(12, 16, 19)
graph.value = 1/3
```

Note: Setting value to -1 will light all LEDs. However, querying it subsequently will return 1 as both representations are the same in hardware. The readable range of `value` is effectively -1 < value <= 1.

values
An infinite iterator of values read from *value*.

PI-TRAFFIC

class gpiozero.**PiTraffic**(*pwm=False*, *initial_value=False*)
Extends `TrafficLights` for the Low Voltage Labs PI-TRAFFIC: vertical traffic lights board when attached to GPIO pins 9, 10, and 11.

There's no need to specify the pins if the PI-TRAFFIC is connected to the default pins (9, 10, 11). The following example turns on the amber LED on the PI-TRAFFIC:

```
from gpiozero import PiTraffic

traffic = PiTraffic()
traffic.amber.on()
```

To use the PI-TRAFFIC board when attached to a non-standard set of pins, simply use the parent class, `TrafficLights`.

Parameters

- **pwm** (*bool*) – If `True`, construct `PWMLED` instances to represent each LED. If `False` (the default), construct regular `LED` instances.
- **initial_value** (*bool*) – If `False` (the default), all LEDs will be off initially. If `None`, each device will be left in whatever state the pin is found in when configured for output (warning: this can be on). If `True`, the device will be switched on initially.

blink (*on_time=1*, *off_time=1*, *fade_in_time=0*, *fade_out_time=0*, *n=None*, *background=True*)
Make all the LEDs turn on and off repeatedly.

Parameters

- **on_time** (*float*) – Number of seconds on. Defaults to 1 second.
- **off_time** (*float*) – Number of seconds off. Defaults to 1 second.
- **fade_in_time** (*float*) – Number of seconds to spend fading in. Defaults to 0. Must be 0 if `pwm` was `False` when the class was constructed (`ValueError` will be raised if not).
- **fade_out_time** (*float*) – Number of seconds to spend fading out. Defaults to 0. Must be 0 if `pwm` was `False` when the class was constructed (`ValueError` will be raised if not).
- **n** (*int*) – Number of times to blink; `None` (the default) means forever.
- **background** (*bool*) – If `True`, start a background thread to continue blinking and return immediately. If `False`, only return when the blink is finished (warning: the default value of *n* will result in this method never returning).

close ()
Shut down the device and release all associated resources. This method can be called on an already closed device without raising an exception.

This method is primarily intended for interactive use at the command line. It disables the device and releases its pin(s) for use by another device.

You can attempt to do this simply by deleting an object, but unless you've cleaned up all references to the object this may not work (even if you've cleaned up all references, there's still no guarantee the garbage collector will actually delete the object at that point). By contrast, the close method provides a means of ensuring that the object is shut down.

For example, if you have a breadboard with a buzzer connected to pin 16, but then wish to attach an LED instead:

```
>>> from gpiozero import *
>>> bz = Buzzer(16)
>>> bz.on()
>>> bz.off()
>>> bz.close()
>>> led = LED(16)
>>> led.blink()
```

`Device` descendents can also be used as context managers using the `with` statement. For example:

```
>>> from gpiozero import *
>>> with Buzzer(16) as bz:
...     bz.on()
...
>>> with LED(16) as led:
...     led.on()
...
```

off (**args*)
Turn all the output devices off.

on (**args*)
Turn all the output devices on.

pulse (*fade_in_time=1*, *fade_out_time=1*, *n=None*, *background=True*)
Make the device fade in and out repeatedly.

Parameters

- **fade_in_time** (*float*) – Number of seconds to spend fading in. Defaults to 1.
- **fade_out_time** (*float*) – Number of seconds to spend fading out. Defaults to 1.
- **n** (*int*) – Number of times to blink; `None` (the default) means forever.
- **background** (*bool*) – If `True` (the default), start a background thread to continue blinking and return immediately. If `False`, only return when the blink is finished (warning: the default value of *n* will result in this method never returning).

toggle(**args*)
Toggle all the output devices. For each device, if it's on, turn it off; if it's off, turn it on.

leds
A flat tuple of all LEDs contained in this collection (and all sub-collections).

source
The iterable to use as a source of values for `value`.

source_delay
The delay (measured in seconds) in the loop used to read values from `source`. Defaults to 0.01 seconds which is generally sufficient to keep CPU usage to a minimum while providing adequate responsiveness.

value
A tuple containing a value for each subordinate device. This property can also be set to update the state of all subordinate output devices.

values
An infinite iterator of values read from *value*.

TrafficLightsBuzzer

class gpiozero.**TrafficLightsBuzzer**(*lights*, *buzzer*, *button*)
Extends `CompositeOutputDevice` and is a generic class for HATs with traffic lights, a button and a buzzer.

Parameters

- **lights** (`TrafficLights`) – An instance of `TrafficLights` representing the traffic lights of the HAT.
- **buzzer** (`Buzzer`) – An instance of `Buzzer` representing the buzzer on the HAT.
- **button** (`Button`) – An instance of `Button` representing the button on the HAT.

off()
Turn all the output devices off.

on()
Turn all the output devices on.

toggle()
Toggle all the output devices. For each device, if it's on, turn it off; if it's off, turn it on.

source
The iterable to use as a source of values for `value`.

source_delay
The delay (measured in seconds) in the loop used to read values from `source`. Defaults to 0.01 seconds which is generally sufficient to keep CPU usage to a minimum while providing adequate responsiveness.

value
A tuple containing a value for each subordinate device. This property can also be set to update the state of all subordinate output devices.

values
An infinite iterator of values read from *value*.

Fish Dish

class gpiozero.**FishDish**(*pwm=False*)
Extends *TrafficLightsBuzzer* for the Pi Supply FishDish: traffic light LEDs, a button and a buzzer.

The FishDish pins are fixed and therefore there's no need to specify them when constructing this class. The following example waits for the button to be pressed on the FishDish, then turns on all the LEDs:

```
from gpiozero import FishDish

fish = FishDish()
fish.button.wait_for_press()
fish.lights.on()
```

Parameters pwm (*bool*) – If True, construct *PWMLED* instances to represent each LED. If False (the default), construct regular *LED* instances.

off()
Turn all the output devices off.

on()
Turn all the output devices on.

toggle()
Toggle all the output devices. For each device, if it's on, turn it off; if it's off, turn it on.

source
The iterable to use as a source of values for *value*.

source_delay
The delay (measured in seconds) in the loop used to read values from *source*. Defaults to 0.01 seconds which is generally sufficient to keep CPU usage to a minimum while providing adequate responsiveness.

value
A tuple containing a value for each subordinate device. This property can also be set to update the state of all subordinate output devices.

values
An infinite iterator of values read from *value*.

Traffic HAT

class gpiozero.**TrafficHat**(*pwm=False*)
Extends *TrafficLightsBuzzer* for the Ryanteck Traffic HAT: traffic light LEDs, a button and a buzzer.

The Traffic HAT pins are fixed and therefore there's no need to specify them when constructing this class. The following example waits for the button to be pressed on the Traffic HAT, then turns on all the LEDs:

```
from gpiozero import TrafficHat

hat = TrafficHat()
hat.button.wait_for_press()
hat.lights.on()
```

Parameters pwm (*bool*) – If True, construct *PWMLED* instances to represent each LED. If False (the default), construct regular *LED* instances.

off()
Turn all the output devices off.

on()
Turn all the output devices on.

toggle()
Toggle all the output devices. For each device, if it's on, turn it off; if it's off, turn it on.

source
The iterable to use as a source of values for `value`.

source_delay
The delay (measured in seconds) in the loop used to read values from `source`. Defaults to 0.01 seconds which is generally sufficient to keep CPU usage to a minimum while providing adequate responsiveness.

value
A tuple containing a value for each subordinate device. This property can also be set to update the state of all subordinate output devices.

values
An infinite iterator of values read from *value*.

Robot

class gpiozero.**Robot**(*left=None*, *right=None*)
Extends `CompositeDevice` to represent a generic dual-motor robot.

This class is constructed with two tuples representing the forward and backward pins of the left and right controllers respectively. For example, if the left motor's controller is connected to GPIOs 4 and 14, while the right motor's controller is connected to GPIOs 17 and 18 then the following example will drive the robot forward:

```
from gpiozero import Robot

robot = Robot(left=(4, 14), right=(17, 18))
robot.forward()
```

Parameters

- **left** (`tuple`) – A tuple of two GPIO pins representing the forward and backward inputs of the left motor's controller.
- **right** (`tuple`) – A tuple of two GPIO pins representing the forward and backward inputs of the right motor's controller.

backward(*speed=1*)
Drive the robot backward by running both motors backward.

Parameters **speed** (`float`) – Speed at which to drive the motors, as a value between 0 (stopped) and 1 (full speed). The default is 1.

forward(*speed=1*)
Drive the robot forward by running both motors forward.

Parameters **speed** (`float`) – Speed at which to drive the motors, as a value between 0 (stopped) and 1 (full speed). The default is 1.

left(*speed=1*)
Make the robot turn left by running the right motor forward and left motor backward.

Parameters **speed** (`float`) – Speed at which to drive the motors, as a value between 0 (stopped) and 1 (full speed). The default is 1.

reverse()
Reverse the robot's current motor directions. If the robot is currently running full speed forward, it will run full speed backward. If the robot is turning left at half-speed, it will turn right at half-speed. If the robot is currently stopped it will remain stopped.

right(*speed=1*)
Make the robot turn right by running the left motor forward and right motor backward.

Parameters **speed** (`float`) – Speed at which to drive the motors, as a value between 0 (stopped) and 1 (full speed). The default is 1.

stop()
Stop the robot.

source
The iterable to use as a source of values for `value`.

source_delay
The delay (measured in seconds) in the loop used to read values from `source`. Defaults to 0.01 seconds which is generally sufficient to keep CPU usage to a minimum while providing adequate responsiveness.

value
Represents the motion of the robot as a tuple of (left_motor_speed, right_motor_speed) with `(-1, -1)` representing full speed backwards, `(1, 1)` representing full speed forwards, and `(0, 0)` representing stopped.

values
An infinite iterator of values read from *value*.

Ryanteck MCB Robot

class gpiozero.**RyanteckRobot**
Extends `Robot` for the Ryanteck MCB robot.

The Ryanteck MCB pins are fixed and therefore there's no need to specify them when constructing this class. The following example drives the robot forward:

```
from gpiozero import RyanteckRobot

robot = RyanteckRobot()
robot.forward()
```

backward(*speed=1*)
Drive the robot backward by running both motors backward.

Parameters **speed** (`float`) – Speed at which to drive the motors, as a value between 0 (stopped) and 1 (full speed). The default is 1.

forward(*speed=1*)
Drive the robot forward by running both motors forward.

Parameters **speed** (`float`) – Speed at which to drive the motors, as a value between 0 (stopped) and 1 (full speed). The default is 1.

left(*speed=1*)
Make the robot turn left by running the right motor forward and left motor backward.

Parameters **speed** (`float`) – Speed at which to drive the motors, as a value between 0 (stopped) and 1 (full speed). The default is 1.

reverse()
Reverse the robot's current motor directions. If the robot is currently running full speed forward, it will run full speed backward. If the robot is turning left at half-speed, it will turn right at half-speed. If the robot is currently stopped it will remain stopped.

right (*speed=1*)
Make the robot turn right by running the left motor forward and right motor backward.

Parameters **speed** (*float*) – Speed at which to drive the motors, as a value between 0 (stopped) and 1 (full speed). The default is 1.

stop ()
Stop the robot.

source
The iterable to use as a source of values for *value*.

source_delay
The delay (measured in seconds) in the loop used to read values from *source*. Defaults to 0.01 seconds which is generally sufficient to keep CPU usage to a minimum while providing adequate responsiveness.

value
Represents the motion of the robot as a tuple of (left_motor_speed, right_motor_speed) with `(-1, -1)` representing full speed backwards, `(1, 1)` representing full speed forwards, and `(0, 0)` representing stopped.

values
An infinite iterator of values read from *value*.

CamJam #3 Kit Robot

class gpiozero.**CamJamKitRobot**
Extends *Robot* for the CamJam #3 EduKit robot controller.

The CamJam robot controller pins are fixed and therefore there's no need to specify them when constructing this class. The following example drives the robot forward:

```
from gpiozero import CamJamKitRobot

robot = CamJamKitRobot()
robot.forward()
```

backward (*speed=1*)
Drive the robot backward by running both motors backward.

Parameters **speed** (*float*) – Speed at which to drive the motors, as a value between 0 (stopped) and 1 (full speed). The default is 1.

forward (*speed=1*)
Drive the robot forward by running both motors forward.

Parameters **speed** (*float*) – Speed at which to drive the motors, as a value between 0 (stopped) and 1 (full speed). The default is 1.

left (*speed=1*)
Make the robot turn left by running the right motor forward and left motor backward.

Parameters **speed** (*float*) – Speed at which to drive the motors, as a value between 0 (stopped) and 1 (full speed). The default is 1.

reverse ()
Reverse the robot's current motor directions. If the robot is currently running full speed forward, it will run full speed backward. If the robot is turning left at half-speed, it will turn right at half-speed. If the robot is currently stopped it will remain stopped.

right (*speed=1*)
Make the robot turn right by running the left motor forward and right motor backward.

Parameters **speed** (*float*) – Speed at which to drive the motors, as a value between 0 (stopped) and 1 (full speed). The default is 1.

stop()
Stop the robot.

source
The iterable to use as a source of values for *value*.

source_delay
The delay (measured in seconds) in the loop used to read values from *source*. Defaults to 0.01 seconds which is generally sufficient to keep CPU usage to a minimum while providing adequate responsiveness.

value
Represents the motion of the robot as a tuple of (left_motor_speed, right_motor_speed) with `(-1, -1)` representing full speed backwards, `(1, 1)` representing full speed forwards, and `(0, 0)` representing stopped.

values
An infinite iterator of values read from *value*.

Energenie

class gpiozero.**Energenie**(*socket=None*, *initial_value=False*)
Extends *Device* to represent an Energenie socket controller.

This class is constructed with a socket number and an optional initial state (defaults to `False`, meaning off). Instances of this class can be used to switch peripherals on and off. For example:

```
from gpiozero import Energenie

lamp = Energenie(1)
lamp.on()
```

Parameters

- **socket** (*int*) – Which socket this instance should control. This is an integer number between 1 and 4.
- **initial_value** (*bool*) – The initial state of the socket. As Energenie sockets provide no means of reading their state, you must provide an initial state for the socket, which will be set upon construction. This defaults to `False` which will switch the socket off.

close()
Shut down the device and release all associated resources. This method can be called on an already closed device without raising an exception.

This method is primarily intended for interactive use at the command line. It disables the device and releases its pin(s) for use by another device.

You can attempt to do this simply by deleting an object, but unless you've cleaned up all references to the object this may not work (even if you've cleaned up all references, there's still no guarantee the garbage collector will actually delete the object at that point). By contrast, the close method provides a means of ensuring that the object is shut down.

For example, if you have a breadboard with a buzzer connected to pin 16, but then wish to attach an LED instead:

```
>>> from gpiozero import *
>>> bz = Buzzer(16)
>>> bz.on()
```

```
>>> bz.off()
>>> bz.close()
>>> led = LED(16)
>>> led.blink()
```

`Device` descendents can also be used as context managers using the `with` statement. For example:

```
>>> from gpiozero import *
>>> with Buzzer(16) as bz:
...     bz.on()
...
>>> with LED(16) as led:
...     led.on()
...
```

is_active
Returns `True` if the device is currently active and `False` otherwise. This property is usually derived from `value`. Unlike `value`, this is *always* a boolean.

source
The iterable to use as a source of values for `value`.

source_delay
The delay (measured in seconds) in the loop used to read values from `source`. Defaults to 0.01 seconds which is generally sufficient to keep CPU usage to a minimum while providing adequate responsiveness.

values
An infinite iterator of values read from *value*.

SnowPi

class gpiozero.**SnowPi**(*pwm=False*, *initial_value=False*)
Extends `LEDBoard` for the Ryanteck SnowPi board.

The SnowPi pins are fixed and therefore there's no need to specify them when constructing this class. The following example turns on the eyes, sets the nose pulsing, and the arms blinking:

```
from gpiozero import SnowPi

snowman = SnowPi(pwm=True)
snowman.eyes.on()
snowman.nose.pulse()
snowman.arms.blink()
```

Parameters

- **pwm** (`bool`) – If `True`, construct `PWMLED` instances to represent each LED. If `False` (the default), construct regular `LED` instances.
- **initial_value** (`bool`) – If `False` (the default), all LEDs will be off initially. If `None`, each device will be left in whatever state the pin is found in when configured for output (warning: this can be on). If `True`, the device will be switched on initially.

blink(*on_time=1*, *off_time=1*, *fade_in_time=0*, *fade_out_time=0*, *n=None*, *background=True*)
Make all the LEDs turn on and off repeatedly.

Parameters

- **on_time** (`float`) – Number of seconds on. Defaults to 1 second.
- **off_time** (`float`) – Number of seconds off. Defaults to 1 second.

- **fade_in_time** (*float*) – Number of seconds to spend fading in. Defaults to 0. Must be 0 if `pwm` was `False` when the class was constructed (`ValueError` will be raised if not).
- **fade_out_time** (*float*) – Number of seconds to spend fading out. Defaults to 0. Must be 0 if `pwm` was `False` when the class was constructed (`ValueError` will be raised if not).
- **n** (*int*) – Number of times to blink; `None` (the default) means forever.
- **background** (*bool*) – If `True`, start a background thread to continue blinking and return immediately. If `False`, only return when the blink is finished (warning: the default value of *n* will result in this method never returning).

close()

Shut down the device and release all associated resources. This method can be called on an already closed device without raising an exception.

This method is primarily intended for interactive use at the command line. It disables the device and releases its pin(s) for use by another device.

You can attempt to do this simply by deleting an object, but unless you've cleaned up all references to the object this may not work (even if you've cleaned up all references, there's still no guarantee the garbage collector will actually delete the object at that point). By contrast, the close method provides a means of ensuring that the object is shut down.

For example, if you have a breadboard with a buzzer connected to pin 16, but then wish to attach an LED instead:

```
>>> from gpiozero import *
>>> bz = Buzzer(16)
>>> bz.on()
>>> bz.off()
>>> bz.close()
>>> led = LED(16)
>>> led.blink()
```

`Device` descendents can also be used as context managers using the `with` statement. For example:

```
>>> from gpiozero import *
>>> with Buzzer(16) as bz:
...     bz.on()
...
>>> with LED(16) as led:
...     led.on()
...
```

off(**args*)

Turn all the output devices off.

on(**args*)

Turn all the output devices on.

pulse(*fade_in_time=1, fade_out_time=1, n=None, background=True*)

Make the device fade in and out repeatedly.

Parameters

- **fade_in_time** (*float*) – Number of seconds to spend fading in. Defaults to 1.
- **fade_out_time** (*float*) – Number of seconds to spend fading out. Defaults to 1.
- **n** (*int*) – Number of times to blink; `None` (the default) means forever.
- **background** (*bool*) – If `True` (the default), start a background thread to continue blinking and return immediately. If `False`, only return when the blink is finished (warning: the default value of *n* will result in this method never returning).

toggle (**args*)
Toggle all the output devices. For each device, if it's on, turn it off; if it's off, turn it on.

leds
A flat tuple of all LEDs contained in this collection (and all sub-collections).

source
The iterable to use as a source of values for `value`.

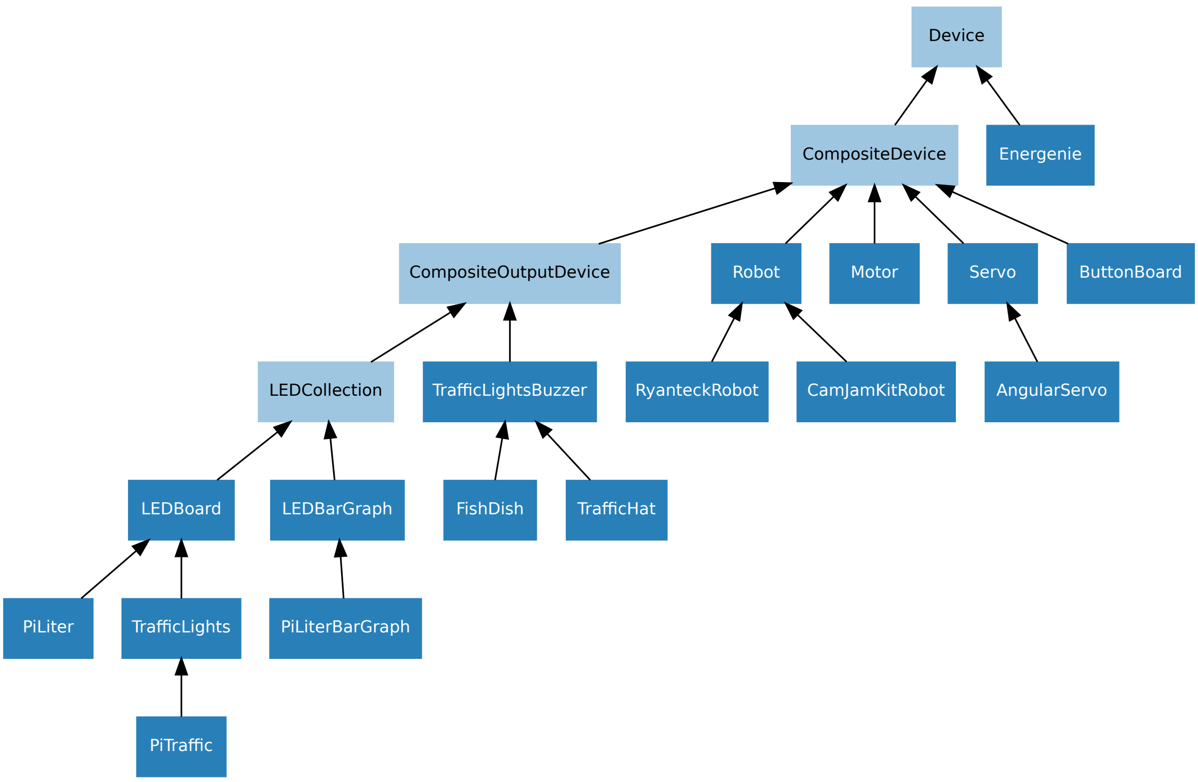
source_delay
The delay (measured in seconds) in the loop used to read values from `source`. Defaults to 0.01 seconds which is generally sufficient to keep CPU usage to a minimum while providing adequate responsiveness.

value
A tuple containing a value for each subordinate device. This property can also be set to update the state of all subordinate output devices.

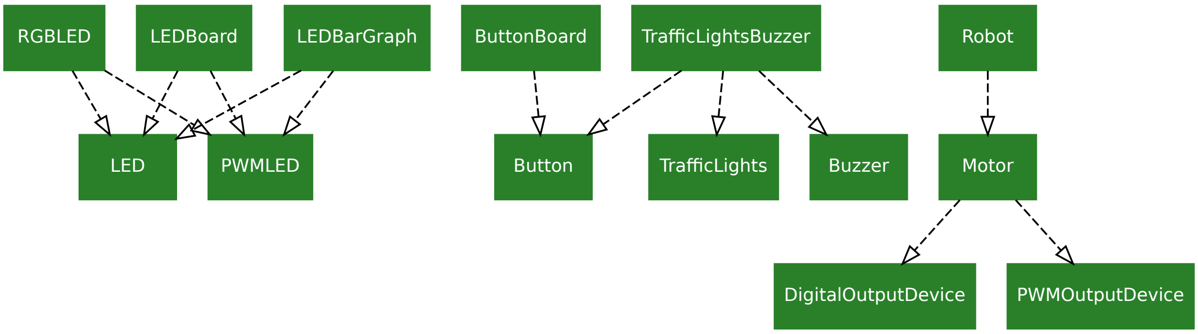
values
An infinite iterator of values read from *value*.

Base Classes

The classes in the sections above are derived from a series of base classes, some of which are effectively abstract. The classes form the (partial) hierarchy displayed in the graph below:



For composite devices, the following chart shows which devices are composed of which other devices:



The following sections document these base classes for advanced users that wish to construct classes for their own devices.

LEDCollection

class gpiozero.**LEDCollection**(**args*, ***kwargs*)

Extends *CompositeOutputDevice*. Abstract base class for *LEDBoard* and *LEDBarGraph*.

leds

A flat tuple of all LEDs contained in this collection (and all sub-collections).

CompositeOutputDevice

class gpiozero.**CompositeOutputDevice**(**args*, *_order=None*, ***kwargs*)

Extends *CompositeDevice* with *on()*, *off()*, and *toggle()* methods for controlling subordinate output devices. Also extends *value* to be writeable.

Parameters **_order** (*list*) – If specified, this is the order of named items specified by keyword arguments (to ensure that the *value* tuple is constructed with a specific order). All keyword arguments *must* be included in the collection. If omitted, an alphabetically sorted order will be selected for keyword arguments.

off()

Turn all the output devices off.

on()

Turn all the output devices on.

toggle()

Toggle all the output devices. For each device, if it's on, turn it off; if it's off, turn it on.

value

A tuple containing a value for each subordinate device. This property can also be set to update the state of all subordinate output devices.

CompositeDevice

class gpiozero.**CompositeDevice**(**args*, *_order=None*, ***kwargs*)

Extends *Device*. Represents a device composed of multiple devices like simple HATs, H-bridge motor controllers, robots composed of multiple motors, etc.

The constructor accepts subordinate devices as positional or keyword arguments. Positional arguments form unnamed devices accessed via the `all` attribute, while keyword arguments are added to the device as named (read-only) attributes.

Parameters **_order** (*list*) – If specified, this is the order of named items specified by keyword arguments (to ensure that the `value` tuple is constructed with a specific order). All keyword arguments *must* be included in the collection. If omitted, an alphabetically sorted order will be selected for keyword arguments.

close()

Shut down the device and release all associated resources. This method can be called on an already closed device without raising an exception.

This method is primarily intended for interactive use at the command line. It disables the device and releases its pin(s) for use by another device.

You can attempt to do this simply by deleting an object, but unless you've cleaned up all references to the object this may not work (even if you've cleaned up all references, there's still no guarantee the garbage collector will actually delete the object at that point). By contrast, the close method provides a means of ensuring that the object is shut down.

For example, if you have a breadboard with a buzzer connected to pin 16, but then wish to attach an LED instead:

```
>>> from gpiozero import *
>>> bz = Buzzer(16)
>>> bz.on()
>>> bz.off()
>>> bz.close()
>>> led = LED(16)
>>> led.blink()
```

`Device` descendents can also be used as context managers using the `with` statement. For example:

```
>>> from gpiozero import *
>>> with Buzzer(16) as bz:
...     bz.on()
...
>>> with LED(16) as led:
...     led.on()
...
```

Internal Devices

GPIO Zero also provides several "internal" devices which represent facilities provided by the operating system itself. These can be used to react to things like the time of day, or whether a server is available on the network.

Warning: These devices are experimental and their API is not yet considered stable. We welcome any comments from testers, especially regarding new "internal devices" that you'd find useful!

TimeOfDay

class gpiozero.**TimeOfDay**(*start_time*, *end_time*, *utc=True*)

Extends `InternalDevice` to provide a device which is active when the computer's clock indicates that the current time is between *start_time* and *end_time* (inclusive) which are `time` instances.

The following example turns on a lamp attached to an `Energenie` plug between 7 and 8 AM:

```
from datetime import time
from gpiozero import TimeOfDay, Energenie
from signal import pause

lamp = Energenie(0)
morning = TimeOfDay(time(7), time(8))
```

```
morning.when_activated = lamp.on
morning.when_deactivated = lamp.off
pause()
```

Parameters

- **start_time** (*time*) – The time from which the device will be considered active.
- **end_time** (*time*) – The time after which the device will be considered inactive.
- **utc** (*bool*) – If `True` (the default), a naive UTC time will be used for the comparison rather than a local time-zone reading.

PingServer

class gpiozero.**PingServer**(*host*)

Extends `InternalDevice` to provide a device which is active when a *host* on the network can be pinged.

The following example lights an LED while a server is reachable (note the use of `source_delay` to ensure the server is not flooded with pings):

```
from gpiozero import PingServer, LED
from signal import pause

server = PingServer('my-server')
led = LED(4)
led.source_delay = 1
led.source = server.values
pause()
```

Parameters **host** (*str*) – The hostname or IP address to attempt to ping.

CPUTemperature

class gpiozero.**CPUTemperature**(*sensor_file='/sys/class/thermal/thermal_zone0/temp'*, *min_temp=0.0*, *max_temp=100.0*, *threshold=80.0*)

Extends `InternalDevice` to provide a device which is active when the CPU temperature exceeds the *threshold* value.

The following example plots the CPU's temperature on an LED bar graph:

```
from gpiozero import LEDBarGraph, CPUTemperature
from signal import pause

# Use minimums and maximums that are closer to "normal" usage so the
# bar graph is a bit more "lively"
temp = CPUTemperature(min_temp=50, max_temp=90)
graph = LEDBarGraph(5, 6, 13, 19, 25, pwm=True)
graph.source = temp.values
pause()
```

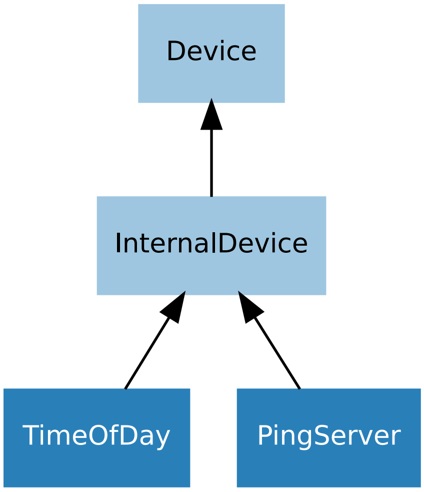
Parameters

- **sensor_file** (*str*) – The file from which to read the temperature. This defaults to the sysfs file `/sys/class/thermal/thermal_zone0/temp`. Whatever file is specified is expected to contain a single line containing the temperature in milli-degrees celsius.
- **min_temp** (*float*) – The temperature at which `value` will read 0.0. This defaults to 0.0.

- **max_temp** (*float*) – The temperature at which `value` will read 1.0. This defaults to 100.0.
- **threshold** (*float*) – The temperature above which the device will be considered "active". This defaults to 80.0.

Base Classes

The classes in the sections above are derived from a series of base classes, some of which are effectively abstract. The classes form the (partial) hierarchy displayed in the graph below:



The following sections document these base classes for advanced users that wish to construct classes for their own devices.

InternalDevice

class `gpiozero.`**`InternalDevice`**

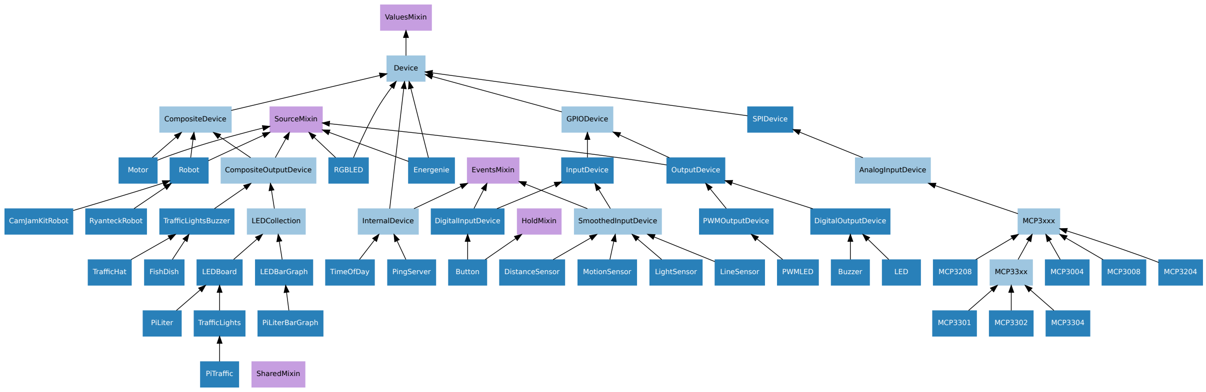
Extends *`Device`* to provide a basis for devices which have no specific hardware representation. These are effectively pseudo-devices and usually represent operating system services like the internal clock, file systems or network facilities.

Generic Classes

The GPIO Zero class hierarchy is quite extensive. It contains several base classes (most of which are documented in their corresponding chapters):

- *`Device`* is the root of the hierarchy, implementing base functionality like *`close()`* and context manager handlers.
- *`GPIODevice`* represents individual devices that attach to a single GPIO pin
- *`SPIDevice`* represents devices that communicate over an SPI interface (implemented as four GPIO pins)
- *`InternalDevice`* represents devices that are entirely internal to the Pi (usually operating system related services)
- *`CompositeDevice`* represents devices composed of multiple other devices like HATs

There are also several mixin classes for adding important functionality at numerous points in the hierarchy, which is illustrated below:



Device

class `gpiozero.`**`Device`**

Represents a single device of any type; GPIO-based, SPI-based, I2C-based, etc. This is the base class of the device hierarchy. It defines the basic services applicable to all devices (specifically thhe *`is_active`* property, the *`value`* property, and the *`close()`* method).

`close``()`

Shut down the device and release all associated resources. This method can be called on an already closed device without raising an exception.

This method is primarily intended for interactive use at the command line. It disables the device and releases its pin(s) for use by another device.

You can attempt to do this simply by deleting an object, but unless you've cleaned up all references to the object this may not work (even if you've cleaned up all references, there's still no guarantee the garbage collector will actually delete the object at that point). By contrast, the close method provides a means of ensuring that the object is shut down.

For example, if you have a breadboard with a buzzer connected to pin 16, but then wish to attach an LED instead:

```
>>> from gpiozero import *
>>> bz = Buzzer(16)
>>> bz.on()
>>> bz.off()
>>> bz.close()
>>> led = LED(16)
>>> led.blink()
```

`Device` descendents can also be used as context managers using the `with` statement. For example:

```
>>> from gpiozero import *
>>> with Buzzer(16) as bz:
...     bz.on()
...
>>> with LED(16) as led:
...     led.on()
...
```

`closed`

Returns `True` if the device is closed (see the *`close()`* method). Once a device is closed you can no longer use any other methods or properties to control or query the device.

`is_active`

Returns `True` if the device is currently active and `False` otherwise. This property is usually derived from *`value`*. Unlike *`value`*, this is *always* a boolean.

value
Returns a value representing the device's state. Frequently, this is a boolean value, or a number between 0 and 1 but some devices use larger ranges (e.g. -1 to +1) and composite devices usually use tuples to return the states of all their subordinate components.

ValuesMixin

class gpiozero.**ValuesMixin**(...)
Adds a *values* property to the class which returns an infinite generator of readings from the value property. There is rarely a need to use this mixin directly as all base classes in GPIO Zero include it.

Note: Use this mixin *first* in the parent class list.

values
An infinite iterator of values read from *value*.

SourceMixin

class gpiozero.**SourceMixin**(...)
Adds a *source* property to the class which, given an iterable, sets value to each member of that iterable until it is exhausted. This mixin is generally included in novel output devices to allow their state to be driven from another device.

Note: Use this mixin *first* in the parent class list.

source
The iterable to use as a source of values for value.

source_delay
The delay (measured in seconds) in the loop used to read values from *source*. Defaults to 0.01 seconds which is generally sufficient to keep CPU usage to a minimum while providing adequate responsiveness.

SharedMixin

class gpiozero.**SharedMixin**(...)
This mixin marks a class as "shared". In this case, the meta-class (GPIOMeta) will use *_shared_key()* to convert the constructor arguments to an immutable key, and will check whether any existing instances match that key. If they do, they will be returned by the constructor instead of a new instance. An internal reference counter is used to determine how many times an instance has been "constructed" in this way.

When close() is called, an internal reference counter will be decremented and the instance will only close when it reaches zero.

classmethod _shared_key(**args*, ***kwargs*)
Given the constructor arguments, returns an immutable key representing the instance. The default simply assumes all positional arguments are immutable.

EventsMixin

class gpiozero.**EventsMixin**(...)
Adds edge-detected *when_activated()* and *when_deactivated()* events to a device based on changes to the *is_active* property common to all devices. Also adds *wait_for_active()* and *wait_for_inactive()* methods for level-waiting.

Note: Note that this mixin provides no means of actually firing its events; call `_fire_events()` in sub-classes when device state changes to trigger the events. This should also be called once at the end of initialization to set initial states.

wait_for_active(*timeout=None*)
Pause the script until the device is activated, or the timeout is reached.

Parameters **timeout** (*float*) – Number of seconds to wait before proceeding. If this is `None` (the default), then wait indefinitely until the device is active.

wait_for_inactive(*timeout=None*)
Pause the script until the device is deactivated, or the timeout is reached.

Parameters **timeout** (*float*) – Number of seconds to wait before proceeding. If this is `None` (the default), then wait indefinitely until the device is inactive.

active_time
The length of time (in seconds) that the device has been active for. When the device is inactive, this is `None`.

inactive_time
The length of time (in seconds) that the device has been inactive for. When the device is active, this is `None`.

when_activated
The function to run when the device changes state from inactive to active.

This can be set to a function which accepts no (mandatory) parameters, or a Python function which accepts a single mandatory parameter (with as many optional parameters as you like). If the function accepts a single mandatory parameter, the device that activated will be passed as that parameter.

Set this property to `None` (the default) to disable the event.

when_deactivated
The function to run when the device changes state from active to inactive.

This can be set to a function which accepts no (mandatory) parameters, or a Python function which accepts a single mandatory parameter (with as many optional parameters as you like). If the function accepts a single mandatory parameter, the device that deactivated will be passed as that parameter.

Set this property to `None` (the default) to disable the event.

HoldMixin

class gpiozero.**HoldMixin**(...)
Extends *EventsMixin* to add the *when_held* event and the machinery to fire that event repeatedly (when *hold_repeat* is `True`) at internals defined by *hold_time*.

held_time
The length of time (in seconds) that the device has been held for. This is counted from the first execution of the *when_held* event rather than when the device activated, in contrast to *active_time*. If the device is not currently held, this is `None`.

hold_repeat
If `True`, *when_held* will be executed repeatedly with *hold_time* seconds between each invocation.

hold_time
The length of time (in seconds) to wait after the device is activated, until executing the *when_held* handler. If *hold_repeat* is True, this is also the length of time between invocations of *when_held*.

is_held
When True, the device has been active for at least *hold_time* seconds.

when_held
The function to run when the device has remained active for *hold_time* seconds.

This can be set to a function which accepts no (mandatory) parameters, or a Python function which accepts a single mandatory parameter (with as many optional parameters as you like). If the function accepts a single mandatory parameter, the device that activated will be passed as that parameter.

Set this property to None (the default) to disable the event.

Source Tools

GPIO Zero includes several utility routines which are intended to be used with the *source* and *values* attributes common to most devices in the library. These utility routines are in the tools module of GPIO Zero and are typically imported as follows:

```
from gpiozero.tools import scaled, negated, all_values
```

Given that *source* and *values* deal with infinite iterators, another excellent source of utilities is the itertools module in the standard library.

> **Warning:** While the devices API is now considered stable and won't change in backwards incompatible ways, the tools API is *not* yet considered stable. It is potentially subject to change in future versions. We welcome any comments from testers!

Single source conversions

gpiozero.tools.**absoluted**(*values*)
Returns *values* with all negative elements negated (so that they're positive). For example:

```
from gpiozero import PWMLED, Motor, MCP3008
from gpiozero.tools import absoluted, scaled
from signal import pause

led = PWMLED(4)
motor = Motor(22, 27)
pot = MCP3008(channel=0)

motor.source = scaled(pot.values, -1, 1)
led.source = absoluted(motor.values)

pause()
```

gpiozero.tools.**booleanized**(*values*, *min_value*, *max_value*, *hysteresis=0*)
Returns True for each item in *values* between *min_value* and *max_value*, and False otherwise. *hysteresis* can optionally be used to add hysteresis which prevents the output value rapidly flipping when the input value is fluctuating near the *min_value* or *max_value* thresholds. For example, to light an LED only when a potentiometer is between 1/4 and 3/4 of its full range:

```
from gpiozero import LED, MCP3008
from gpiozero.tools import booleanized
from signal import pause

led = LED(4)
pot = MCP3008(channel=0)
```

```
led.source = booleanized(pot.values, 0.25, 0.75)
pause()
```

gpiozero.tools.**clamped**(*values*, *output_min=0*, *output_max=1*)

Returns *values* clamped from *output_min* to *output_max*, i.e. any items less than *output_min* will be returned as *output_min* and any items larger than *output_max* will be returned as *output_max* (these default to 0 and 1 respectively). For example:

```
from gpiozero import PWMLED, MCP3008
from gpiozero.tools import clamped
from signal import pause

led = PWMLED(4)
pot = MCP3008(channel=0)

led.source = clamped(pot.values, 0.5, 1.0)

pause()
```

gpiozero.tools.**inverted**(*values*, *input_min=0*, *input_max=1*)

Returns the inversion of the supplied values (*input_min* becomes *input_max*, *input_max* becomes *input_min*, *input_min + 0.1* becomes *input_max - 0.1*, etc.). All items in *values* are assumed to be between *input_min* and *input_max* (which default to 0 and 1 respectively), and the output will be in the same range. For example:

```
from gpiozero import MCP3008, PWMLED
from gpiozero.tools import inverted
from signal import pause

led = PWMLED(4)
pot = MCP3008(channel=0)
led.source = inverted(pot.values)
pause()
```

gpiozero.tools.**negated**(*values*)

Returns the negation of the supplied values (`True` becomes `False`, and `False` becomes `True`). For example:

```
from gpiozero import Button, LED
from gpiozero.tools import negated
from signal import pause

led = LED(4)
btn = Button(17)
led.source = negated(btn.values)
pause()
```

gpiozero.tools.**post_delayed**(*values*, *delay*)

Waits for *delay* seconds after returning each item from *values*.

gpiozero.tools.**post_periodic_filtered**(*values*, *repeat_after*, *block*)

After every *repeat_after* items, blocks the next *block* items from *values*. Note that unlike `pre_periodic_filtered()`, *repeat_after* can't be 0. For example, to block every tenth item read from an ADC:

```
from gpiozero import MCP3008
from gpiozero.tools import post_periodic_filtered

adc = MCP3008(channel=0)

for value in post_periodic_filtered(adc.values, 9, 1):
    print(value)
```

`gpiozero.tools.pre_delayed(values, delay)`

Waits for *delay* seconds before returning each item from *values*.

`gpiozero.tools.pre_periodic_filtered(values, block, repeat_after)`

Blocks the first *block* items from *values*, repeating the block after every *repeat_after* items, if *repeat_after* is non-zero. For example, to discard the first 50 values read from an ADC:

```
from gpiozero import MCP3008
from gpiozero.tools import pre_periodic_filtered

adc = MCP3008(channel=0)

for value in pre_periodic_filtered(adc.values, 50, 0):
    print(value)
```

Or to only display every even item read from an ADC:

```
from gpiozero import MCP3008
from gpiozero.tools import pre_periodic_filtered

adc = MCP3008(channel=0)

for value in pre_periodic_filtered(adc.values, 1, 1):
    print(value)
```

`gpiozero.tools.quantized(values, steps, input_min=0, input_max=1)`

Returns *values* quantized to *steps* increments. All items in *values* are assumed to be between *input_min* and *input_max* (which default to 0 and 1 respectively), and the output will be in the same range.

For example, to quantize values between 0 and 1 to 5 "steps" (0.0, 0.25, 0.5, 0.75, 1.0):

```
from gpiozero import PWMLED, MCP3008
from gpiozero.tools import quantized
from signal import pause

led = PWMLED(4)
pot = MCP3008(channel=0)
led.source = quantized(pot.values, 4)
pause()
```

`gpiozero.tools.queued(values, qsize)`

Queues up readings from *values* (the number of readings queued is determined by *qsize*) and begins yielding values only when the queue is full. For example, to "cascade" values along a sequence of LEDs:

```
from gpiozero import LEDBoard, Button
from gpiozero.tools import queued
from signal import pause

leds = LEDBoard(5, 6, 13, 19, 26)
btn = Button(17)

for i in range(4):
    leds[i].source = queued(leds[i + 1].values, 5)
    leds[i].source_delay = 0.01

leds[4].source = btn.values

pause()
```

`gpiozero.tools.smoothed(values, qsize, average=<function mean>)`

Queues up readings from *values* (the number of readings queued is determined by *qsize*) and begins yielding the *average* of the last *qsize* values when the queue is full. The larger the *qsize*, the more the values are smoothed. For example, to smooth the analog values read from an ADC:

```
from gpiozero import MCP3008
from gpiozero.tools import smoothed

adc = MCP3008(channel=0)

for value in smoothed(adc.values, 5):
    print(value)
```

gpiozero.tools.**scaled**(*values*, *output_min*, *output_max*, *input_min=0*, *input_max=1*)

Returns *values* scaled from *output_min* to *output_max*, assuming that all items in *values* lie between *input_min* and *input_max* (which default to 0 and 1 respectively). For example, to control the direction of a motor (which is represented as a value between -1 and 1) using a potentiometer (which typically provides values between 0 and 1):

```
from gpiozero import Motor, MCP3008
from gpiozero.tools import scaled
from signal import pause

motor = Motor(20, 21)
pot = MCP3008(channel=0)
motor.source = scaled(pot.values, -1, 1)
pause()
```

> **Warning:** If *values* contains elements that lie outside *input_min* to *input_max* (inclusive) then the function will not produce values that lie within *output_min* to *output_max* (inclusive).

Combining sources

gpiozero.tools.**all_values**(**values*)

Returns the logical conjunction of all supplied values (the result is only `True` if and only if all input values are simultaneously `True`). One or more *values* can be specified. For example, to light an `LED` only when *both* buttons are pressed:

```
from gpiozero import LED, Button
from gpiozero.tools import all_values
from signal import pause

led = LED(4)
btn1 = Button(20)
btn2 = Button(21)
led.source = all_values(btn1.values, btn2.values)
pause()
```

gpiozero.tools.**any_values**(**values*)

Returns the logical disjunction of all supplied values (the result is `True` if any of the input values are currently `True`). One or more *values* can be specified. For example, to light an `LED` when *any* button is pressed:

```
from gpiozero import LED, Button
from gpiozero.tools import any_values
from signal import pause

led = LED(4)
btn1 = Button(20)
btn2 = Button(21)
```

```
led.source = any_values(btn1.values, btn2.values)
pause()
```

gpiozero.tools.**averaged**(**values*)

Returns the mean of all supplied values. One or more *values* can be specified. For example, to light a `PWMLED` as the average of several potentiometers connected to an `MCP3008` ADC:

```
from gpiozero import MCP3008, PWMLED
from gpiozero.tools import averaged
from signal import pause

pot1 = MCP3008(channel=0)
pot2 = MCP3008(channel=1)
pot3 = MCP3008(channel=2)
led = PWMLED(4)

led.source = averaged(pot1.values, pot2.values, pot3.values)

pause()
```

gpiozero.tools.**multiplied**(**values*)

Returns the product of all supplied values. One or more *values* can be specified. For example, to light a `PWMLED` as the product (i.e. multiplication) of several potentiometers connected to an `MCP3008` ADC:

```
from gpiozero import MCP3008, PWMLED
from gpiozero.tools import multiplied
from signal import pause

pot1 = MCP3008(channel=0)
pot2 = MCP3008(channel=1)
pot3 = MCP3008(channel=2)
led = PWMLED(4)

led.source = multiplied(pot1.values, pot2.values, pot3.values)

pause()
```

gpiozero.tools.**summed**(**values*)

Returns the sum of all supplied values. One or more *values* can be specified. For example, to light a `PWMLED` as the (scaled) sum of several potentiometers connected to an `MCP3008` ADC:

```
from gpiozero import MCP3008, PWMLED
from gpiozero.tools import summed, scaled
from signal import pause

pot1 = MCP3008(channel=0)
pot2 = MCP3008(channel=1)
pot3 = MCP3008(channel=2)
led = PWMLED(4)

led.source = scaled(summed(pot1.values, pot2.values, pot3.values), 0, 1, 0, 3)

pause()
```

Artificial sources

gpiozero.tools.**cos_values**(*period=360*)

Provides an infinite source of values representing a cosine wave (from -1 to +1) which repeats every *period* values. For example, to produce a “siren” effect with a couple of LEDs that repeats once a second:

```
from gpiozero import PWMLED
from gpiozero.tools import cos_values, scaled, inverted
from signal import pause

red = PWMLED(2)
blue = PWMLED(3)

red.source_delay = 0.01
blue.source_delay = red.source_delay
red.source = scaled(cos_values(100), 0, 1, -1, 1)
blue.source = inverted(red.values)

pause()
```

If you require a different range than -1 to +1, see `scaled()`.

`gpiozero.tools.`**`random_values`**`()`

Provides an infinite source of random values between 0 and 1. For example, to produce a "flickering candle" effect with an LED:

```
from gpiozero import PWMLED
from gpiozero.tools import random_values
from signal import pause

led = PWMLED(4)

led.source = random_values()

pause()
```

If you require a wider range than 0 to 1, see `scaled()`.

`gpiozero.tools.`**`sin_values`**`(`*period=360*`)`

Provides an infinite source of values representing a sine wave (from -1 to +1) which repeats every *period* values. For example, to produce a "siren" effect with a couple of LEDs that repeats once a second:

```
from gpiozero import PWMLED
from gpiozero.tools import sin_values, scaled, inverted
from signal import pause

red = PWMLED(2)
blue = PWMLED(3)

red.source_delay = 0.01
blue.source_delay = red.source_delay
red.source = scaled(sin_values(100), 0, 1, -1, 1)
blue.source = inverted(red.values)

pause()
```

If you require a different range than -1 to +1, see `scaled()`.

Pins

As of release 1.1, the GPIO Zero library can be roughly divided into two things: pins and the devices that are connected to them. The majority of the documentation focuses on devices as pins are below the level that most users are concerned with. However, some users may wish to take advantage of the capabilities of alternative GPIO implementations or (in future) use GPIO extender chips. This is the purpose of the pins portion of the library.

When you construct a device, you pass in a GPIO pin number. However, what the library actually expects is a `Pin` implementation. If it finds a simple integer number instead, it uses one of the following classes to provide the *Pin* implementation (classes are listed in favoured order):

1. *gpiozero.pins.rpigpio.RPiGPIOPin*
2. *gpiozero.pins.rpio.RPIOPin*
3. *gpiozero.pins.pigpiod.PiGPIOPin*
4. *gpiozero.pins.native.NativePin*

You can change the default pin implementation by over-writing the `pin_factory` global in the `devices` module like so:

```
from gpiozero.pins.native import NativePin
import gpiozero.devices
# Force the default pin implementation to be NativePin
gpiozero.devices.pin_factory = NativePin

from gpiozero import LED

# This will now use NativePin instead of RPiGPIOPin
led = LED(16)
```

`pin_factory` is a concrete descendent of the abstract *Pin* class. The descendent may take additional parameters in its constructor provided they are optional; GPIO Zero will expect to be able to construct instances with nothing more than an integer pin number.

However, the descendent may take default information from additional sources. For example, to default to creating pins with *gpiozero.pins.pigpiod.PiGPIOPin* on a remote pi called `remote-pi` you can set the `PIGPIO_ADDR` environment variable when running your script:

```
$ PIGPIO_ADDR=remote-pi python my_script.py
```

It is worth noting that instead of passing an integer to device constructors, you can pass an object derived from *Pin* itself:

```
from gpiozero.pins.native import NativePin
from gpiozero import LED

led = LED(NativePin(16))
```

In future, this separation of pins and devices should also permit the library to utilize pins that are part of IO extender chips. For example:

```
from gpiozero import IOExtender, LED

ext = IOExtender()
led = LED(ext.pins[0])
led.on()
```

Warning: While the devices API is now considered stable and won't change in backwards incompatible ways, the pins API is *not* yet considered stable. It is potentially subject to change in future versions. We welcome any comments from testers!

Warning: The astute and mischievous reader may note that it is possible to mix pin implementations, e.g. using `RPiGPIOPin` for one pin, and `NativePin` for another. This is unsupported, and if it results in your script crashing, your components failing, or your Raspberry Pi turning into an actual raspberry pie, you have only yourself to blame.

RPiGPIOPin

class `gpiozero.pins.rpigpio.`**`RPiGPIOPin`**

Uses the RPi.GPIO library to interface to the Pi's GPIO pins. This is the default pin implementation if the RPi.GPIO library is installed. Supports all features including PWM (via software).

Because this is the default pin implementation you can use it simply by specifying an integer number for the pin in most operations, e.g.:

```
from gpiozero import LED

led = LED(12)
```

However, you can also construct RPi.GPIO pins manually if you wish:

```
from gpiozero.pins.rpigpio import RPiGPIOPin
from gpiozero import LED

led = LED(RPiGPIOPin(12))
```

RPIOPin

class `gpiozero.pins.rpio.`**`RPIOPin`**

Uses the RPIO library to interface to the Pi's GPIO pins. This is the default pin implementation if the RPi.GPIO library is not installed, but RPIO is. Supports all features including PWM (hardware via DMA).

Note: Please note that at the time of writing, RPIO is only compatible with Pi 1's; the Raspberry Pi 2 Model B is *not* supported. Also note that root access is required so scripts must typically be run with `sudo`.

You can construct RPIO pins manually like so:

```
from gpiozero.pins.rpio import RPIOPin
from gpiozero import LED

led = LED(RPIOPin(12))
```

PiGPIOPin

class `gpiozero.pins.pigpiod.`**`PiGPIOPin`**

Uses the pigpio library to interface to the Pi's GPIO pins. The pigpio library relies on a daemon (`pigpiod`) to be running as root to provide access to the GPIO pins, and communicates with this daemon over a network socket.

While this does mean only the daemon itself should control the pins, the architecture does have several advantages:

- Pins can be remote controlled from another machine (the other machine doesn't even have to be a Raspberry Pi; it simply needs the pigpio client library installed on it)
- The daemon supports hardware PWM via the DMA controller
- Your script itself doesn't require root privileges; it just needs to be able to communicate with the daemon

You can construct pigpiod pins manually like so:

```
from gpiozero.pins.pigpiod import PiGPIOPin
from gpiozero import LED
```

```
led = LED(PiGPIOPin(12))
```

This is particularly useful for controlling pins on a remote machine. To accomplish this simply specify the host (and optionally port) when constructing the pin:

```
from gpiozero.pins.pigpiod import PiGPIOPin
from gpiozero import LED
from signal import pause

led = LED(PiGPIOPin(12, host='192.168.0.2'))
```

Note: In some circumstances, especially when playing with PWM, it does appear to be possible to get the daemon into "unusual" states. We would be most interested to hear any bug reports relating to this (it may be a bug in our pin implementation). A workaround for now is simply to restart the `pigpiod` daemon.

NativePin

class `gpiozero.pins.native.`**`NativePin`**

Uses a built-in pure Python implementation to interface to the Pi's GPIO pins. This is the default pin implementation if no third-party libraries are discovered.

> **Warning:** This implementation does *not* currently support PWM. Attempting to use any class which requests PWM will raise an exception. This implementation is also experimental; we make no guarantees it will not eat your Pi for breakfast!

You can construct native pin instances manually like so:

```
from gpiozero.pins.native import NativePin
from gpiozero import LED

led = LED(NativePin(12))
```

Abstract Pin

class `gpiozero.`**`Pin`**

Abstract base class representing a GPIO pin or a pin from an IO extender.

Descendents should override property getters and setters to accurately represent the capabilities of pins. The following functions *must* be overridden:

- `_get_function()`
- `_set_function()`
- `_get_state()`

The following functions *may* be overridden if applicable:

- *`close()`*
- `_set_state()`
- `_get_frequency()`
- `_set_frequency()`
- `_get_pull()`

- _set_pull()
- _get_bounce()
- _set_bounce()
- _get_edges()
- _set_edges()
- _get_when_changed()
- _set_when_changed()
- *pi_info()*
- *output_with_state()*
- *input_with_pull()*

> **Warning:** Descendents must ensure that pin instances representing the same physical hardware are identical, right down to object identity. The framework relies on this to correctly clean up resources at interpreter shutdown.

close()

Cleans up the resources allocated to the pin. After this method is called, this *Pin* instance may no longer be used to query or control the pin's state.

input_with_pull(*pull*)

Sets the pin's function to "input" and specifies an initial pull-up for the pin. By default this is equivalent to performing:

```
pin.function = 'input'
pin.pull = pull
```

However, descendents may override this order to provide the smallest possible delay between configuring the pin for input and pulling the pin up/down (which can be important for avoiding "blips" in some configurations).

output_with_state(*state*)

Sets the pin's function to "output" and specifies an initial state for the pin. By default this is equivalent to performing:

```
pin.function = 'output'
pin.state = state
```

However, descendents may override this in order to provide the smallest possible delay between configuring the pin for output and specifying an initial value (which can be important for avoiding "blips" in active-low configurations).

classmethod pi_info()

Returns a *PiBoardInfo* instance representing the Pi that instances of this pin class will be attached to.

If the pins represented by this class are not *directly* attached to a Pi (e.g. the pin is attached to a board attached to the Pi, or the pins are not on a Pi at all), this may return `None`.

bounce

The amount of bounce detection (elimination) currently in use by edge detection, measured in seconds. If bounce detection is not currently in use, this is `None`.

If the pin does not support edge detection, attempts to set this property will raise *PinEdgeDetectUnsupported*. If the pin supports edge detection, the class must implement bounce detection, even if only in software.

edges
The edge that will trigger execution of the function or bound method assigned to *when_changed*. This can be one of the strings "both" (the default), "rising", "falling", or "none".

If the pin does not support edge detection, attempts to set this property will raise *PinEdgeDetectUnsupported*.

frequency
The frequency (in Hz) for the pin's PWM implementation, or `None` if PWM is not currently in use. This value always defaults to `None` and may be changed with certain pin types to activate or deactivate PWM.

If the pin does not support PWM, *PinPWMUnsupported* will be raised when attempting to set this to a value other than `None`.

function
The function of the pin. This property is a string indicating the current function or purpose of the pin. Typically this is the string "input" or "output". However, in some circumstances it can be other strings indicating non-GPIO related functionality.

With certain pin types (e.g. GPIO pins), this attribute can be changed to configure the function of a pin. If an invalid function is specified, for this attribute, *PinInvalidFunction* will be raised.

pull
The pull-up state of the pin represented as a string. This is typically one of the strings "up", "down", or "floating" but additional values may be supported by the underlying hardware.

If the pin does not support changing pull-up state (for example because of a fixed pull-up resistor), attempts to set this property will raise *PinFixedPull*. If the specified value is not supported by the underlying hardware, *PinInvalidPull* is raised.

state
The state of the pin. This is 0 for low, and 1 for high. As a low level view of the pin, no swapping is performed in the case of pull ups (see *pull* for more information).

If PWM is currently active (when *frequency* is not `None`), this represents the PWM duty cycle as a value between 0.0 and 1.0.

If a pin is currently configured for input, and an attempt is made to set this attribute, *PinSetInput* will be raised. If an invalid value is specified for this attribute, *PinInvalidState* will be raised.

when_changed
A function or bound method to be called when the pin's state changes (more specifically when the edge specified by *edges* is detected on the pin). The function or bound method must take no parameters.

If the pin does not support edge detection, attempts to set this property will raise *PinEdgeDetectUnsupported*.

Local Pin

class `gpiozero.`**LocalPin**
Abstract base class representing pins attached locally to a Pi. This forms the base class for local-only pin interfaces (`RPiGPIOPin`, `RPIOPin`, and `NativePin`).

classmethod pi_info()
Returns a *PiBoardInfo* instance representing the local Pi. The Pi's revision is determined by reading `/proc/cpuinfo`. If no valid revision is found, returns `None`.

Utilities

The pins module also contains a database of information about the various revisions of Raspberry Pi. This is used internally to raise warnings when non-physical pins are used, or to raise exceptions when pull-downs are requested on pins with physical pull-up resistors attached. The following functions and classes can be used to query this database:

`gpiozero.`**`pi_info`**`(`*revision=None*`)`
Returns a *`PiBoardInfo`* instance containing information about a *revision* of the Raspberry Pi.

Parameters **`revision`** (*`str`*) – The revision of the Pi to return information about. If this is omitted or `None` (the default), then the library will attempt to determine the model of Pi it is running on and return information about that.

class `gpiozero.`**`PiBoardInfo`**
This class is a `namedtuple()` derivative used to represent information about a particular model of Raspberry Pi. While it is a tuple, it is strongly recommended that you use the following named attributes to access the data contained within.

`physical_pin``(`*function*`)`
Return the physical pin supporting the specified *function*. If no pins support the desired *function*, this function raises *`PinNoPins`*. If multiple pins support the desired *function*, *`PinMultiplePins`* will be raised (use *`physical_pins()`* if you expect multiple pins in the result, such as for electrical ground).

Parameters **`function`** (*`str`*) – The pin function you wish to search for. Usually this is something like "GPIO9" for Broadcom GPIO pin 9.

`physical_pins``(`*function*`)`
Return the physical pins supporting the specified *function* as tuples of `(header, pin_number)` where *header* is a string specifying the header containing the *pin_number*. Note that the return value is a `set` which is not indexable. Use *`physical_pin()`* if you are expecting a single return value.

Parameters **`function`** (*`str`*) – The pin function you wish to search for. Usually this is something like "GPIO9" for Broadcom GPIO pin 9, or "GND" for all the pins connecting to electrical ground.

`pulled_up``(`*function*`)`
Returns a bool indicating whether a physical pull-up is attached to the pin supporting the specified *function*. Either *`PinNoPins`* or *`PinMultiplePins`* may be raised if the function is not associated with a single pin.

Parameters **`function`** (*`str`*) – The pin function you wish to determine pull-up for. Usually this is something like "GPIO9" for Broadcom GPIO pin 9.

`revision`
A string indicating the revision of the Pi. This is unique to each revision and can be considered the "key" from which all other attributes are derived. However, in itself the string is fairly meaningless.

`model`
A string containing the model of the Pi (for example, "B", "B+", "A+", "2B", "CM" (for the Compute Module), or "Zero").

`pcb_revision`
A string containing the PCB revision number which is silk-screened onto the Pi (on some models).

Note: This is primarily useful to distinguish between the model B revision 1.0 and 2.0 (not to be confused with the model 2B) which had slightly different pinouts on their 26-pin GPIO headers.

`released`
A string containing an approximate release date for this revision of the Pi (formatted as yyyyQq, e.g. 2012Q1 means the first quarter of 2012).

`soc`
A string indicating the SoC (system on a chip) that this revision of the Pi is based upon.

`manufacturer`
A string indicating the name of the manufacturer (usually "Sony" but a few others exist).

memory
An integer indicating the amount of memory (in Mb) connected to the SoC.

Note: This can differ substantially from the amount of RAM available to the operating system as the GPU's memory is shared with the CPU. When the camera module is activated, at least 128Mb of RAM is typically reserved for the GPU.

---

storage
A string indicating the type of bootable storage used with this revision of Pi, e.g. "SD", "MicroSD", or "eMMC" (for the Compute Module).

usb
An integer indicating how many USB ports are physically present on this revision of the Pi.

Note: This does *not* include the micro-USB port used to power the Pi.

---

ethernet
An integer indicating how many Ethernet ports are physically present on this revision of the Pi.

wifi
A bool indicating whether this revision of the Pi has wifi built-in.

bluetooth
A bool indicating whether this revision of the Pi has bluetooth built-in.

csi
An integer indicating the number of CSI (camera) ports available on this revision of the Pi.

dsi
An integer indicating the number of DSI (display) ports available on this revision of the Pi.

headers
A dictionary which maps header labels to dictionaries which map physical pin numbers to *PinInfo* tuples. For example, to obtain information about pin 12 on header P1 you would query `headers['P1'][12]`.

class gpiozero.**PinInfo**
This class is a `namedtuple()` derivative used to represent information about a pin present on a GPIO header. The following attributes are defined:

number
An integer containing the physical pin number on the header (starting from 1 in accordance with convention).

function
A string describing the function of the pin. Some common examples include "GND" (for pins connecting to ground), "3V3" (for pins which output 3.3 volts), "GPIO9" (for GPIO9 in the Broadcom numbering scheme), etc.

pull_up
A bool indicating whether the pin has a physical pull-up resistor permanently attached (this is usually `False` but GPIO2 and GPIO3 are *usually* `True`). This is used internally by gpiozero to raise errors when pull-down is requested on a pin with a physical pull-up resistor.

Exceptions

The following exceptions are defined by GPIO Zero. Please note that multiple inheritance is heavily used in the exception hierarchy to make testing for exceptions easier. For example, to capture any exception generated by GPIO Zero's code:

```
from gpiozero import *

led = PWMLED(17)
try:
    led.value = 2
except GPIOZeroError:
    print('A GPIO Zero error occurred')
```

Since all GPIO Zero's exceptions descend from *GPIOZeroError*, this will work. However, certain specific errors have multiple parents. For example, in the case that an out of range value is passed to *OutputDevice.value* you would expect a `ValueError` to be raised. In fact, a *OutputDeviceBadValue* error will be raised. However, note that this descends from both *GPIOZeroError* (indirectly) and from `ValueError` so you can still do:

```
from gpiozero import *

led = PWMLED(17)
try:
    led.value = 2
except ValueError:
    print('Bad value specified')
```

Errors

exception gpiozero.**GPIOZeroError**
Base class for all exceptions in GPIO Zero

exception gpiozero.**DeviceClosed**
Error raised when an operation is attempted on a closed device

exception gpiozero.**BadEventHandler**
Error raised when an event handler with an incompatible prototype is specified

exception gpiozero.**BadQueueLen**
Error raised when non-positive queue length is specified

exception gpiozero.**BadWaitTime**
Error raised when an invalid wait time is specified

exception gpiozero.**CompositeDeviceError**
Base class for errors specific to the CompositeDevice hierarchy

exception gpiozero.**CompositeDeviceBadName**
Error raised when a composite device is constructed with a reserved name

exception gpiozero.**EnergenieSocketMissing**
Error raised when socket number is not specified

exception gpiozero.**EnergenieBadSocket**
Error raised when an invalid socket number is passed to *Energenie*

exception gpiozero.**SPIError**
Base class for errors related to the SPI implementation

exception gpiozero.**SPIBadArgs**
Error raised when invalid arguments are given while constructing *SPIDevice*

exception gpiozero.**GPIODeviceError**
Base class for errors specific to the GPIODevice hierarchy

exception gpiozero.**GPIODeviceClosed**
Deprecated descendent of *DeviceClosed*

exception `gpiozero.`**`GPIOPinInUse`**
Error raised when attempting to use a pin already in use by another device

exception `gpiozero.`**`GPIOPinMissing`**
Error raised when a pin number is not specified

exception `gpiozero.`**`InputDeviceError`**
Base class for errors specific to the InputDevice hierarchy

exception `gpiozero.`**`OutputDeviceError`**
Base class for errors specified to the OutputDevice hierarchy

exception `gpiozero.`**`OutputDeviceBadValue`**
Error raised when `value` is set to an invalid value

exception `gpiozero.`**`PinError`**
Base class for errors related to pin implementations

exception `gpiozero.`**`PinInvalidFunction`**
Error raised when attempting to change the function of a pin to an invalid value

exception `gpiozero.`**`PinInvalidState`**
Error raised when attempting to assign an invalid state to a pin

exception `gpiozero.`**`PinInvalidPull`**
Error raised when attempting to assign an invalid pull-up to a pin

exception `gpiozero.`**`PinInvalidEdges`**
Error raised when attempting to assign an invalid edge detection to a pin

exception `gpiozero.`**`PinSetInput`**
Error raised when attempting to set a read-only pin

exception `gpiozero.`**`PinFixedPull`**
Error raised when attempting to set the pull of a pin with fixed pull-up

exception `gpiozero.`**`PinEdgeDetectUnsupported`**
Error raised when attempting to use edge detection on unsupported pins

exception `gpiozero.`**`PinPWMError`**
Base class for errors related to PWM implementations

exception `gpiozero.`**`PinPWMUnsupported`**
Error raised when attempting to activate PWM on unsupported pins

exception `gpiozero.`**`PinPWMFixedValue`**
Error raised when attempting to initialize PWM on an input pin

exception `gpiozero.`**`PinMultiplePins`**
Error raised when multiple pins support the requested function

exception `gpiozero.`**`PinNoPins`**
Error raised when no pins support the requested function

exception `gpiozero.`**`PinUnknownPi`**
Error raised when gpiozero doesn't recognize a revision of the Pi

Warnings

exception `gpiozero.`**`GPIOZeroWarning`**
Base class for all warnings in GPIO Zero

exception `gpiozero.`**`SPIWarning`**
Base class for warnings related to the SPI implementation

exception `gpiozero.`**`SPISoftwareFallback`**
Warning raised when falling back to the software implementation

Changelog

Release 1.3.1 (2016-08-31 ... later)

- Fixed hardware SPI support which Dave broke in 1.3.0. Sorry!
- Some minor docs changes

Release 1.3.0 (2016-08-31)

- Added `ButtonBoard` for reading multiple buttons in a single class (#340)
- Added `Servo` and `AngularServo` classes for controlling simple servo motors (#248)
- Lots of work on supporting easier use of internal and third-party pin implementations (#359)
- `Robot` now has a proper `value` attribute (#305)
- Added `CPUTemperature` as another demo of "internal" devices (#294)
- A temporary work-around for an issue with `DistanceSensor` was included but a full fix is in the works (#385)
- More work on the documentation (#320, #295, #289, etc.)

Not quite as much as we'd hoped to get done this time, but we're rushing to make a Raspbian freeze. As always, thanks to the community - your suggestions and PRs have been brilliant and even if we don't take stuff exactly as is, it's always great to see your ideas. Onto 1.4!

Release 1.2.0 (2016-04-10)

- Added `Energenie` class for controlling Energenie plugs (#69)
- Added `LineSensor` class for single line-sensors (#109)
- Added `DistanceSensor` class for HC-SR04 ultra-sonic sensors (#114)
- Added `SnowPi` class for the Ryanteck Snow-pi board (#130)
- Added `when_held` (and related properties) to `Button` (#115)
- Fixed issues with installing GPIO Zero for python 3 on Raspbian Wheezy releases (#140)
- Added support for lots of ADC chips (MCP3xxx family) (#162) - many thanks to pcopa and lurch!
- Added support for pigpiod as a pin implementation with `PiGPIOPin` (#180)
- Many refinements to the base classes mean more consistency in composite devices and several bugs squashed (#164, #175, #182, #189, #193, #229)
- GPIO Zero is now aware of what sort of Pi it's running on via `pi_info()` and has a fairly extensive database of Pi information which it uses to determine when users request impossible things (like pull-down on a pin with a physical pull-up resistor) (#222)
- The source/values system was enhanced to ensure normal usage doesn't stress the CPU and lots of utilities were added (#181, #251)

And I'll just add a note of thanks to the many people in the community who contributed to this release: we've had some great PRs, suggestions, and bug reports in this version. Of particular note:

- Schelto van Doorn was instrumental in adding support for numerous ADC chips
- Alex Eames generously donated a RasPiO Analog board which was extremely useful in developing the software SPI interface (and testing the ADC support)
- Andrew Scheller squashed several dozen bugs (usually a day or so after Dave had introduced them ;)

As always, many thanks to the whole community - we look forward to hearing from you more in 1.3!

Release 1.1.0 (2016-02-08)

- Documentation converted to reST and expanded to include generic classes and several more recipes (#80, #82, #101, #119, #135, #168)
- New `CamJamKitRobot` class with the pre-defined motor pins for the new CamJam EduKit
- New `LEDBarGraph` class (many thanks to Martin O'Hanlon!) (#126, #176)
- New `Pin` implementation abstracts out the concept of a GPIO pin paving the way for alternate library support and IO extenders in future (#141)
- New `LEDBoard.blink()` method which works properly even when background is set to `False` (#94, #161)
- New `RGBLED.blink()` method which implements (rudimentary) color fading too! (#135, #174)
- New `initial_value` attribute on `OutputDevice` ensures consistent behaviour on construction (#118)
- New `active_high` attribute on `PWMOutputDevice` and `RGBLED` allows use of common anode devices (#143, #154)
- Loads of new ADC chips supported (many thanks to GitHub user pcopa!) (#150)

Release 1.0.0 (2015-11-16)

- Debian packaging added (#44)
- `PWMLED` class added (#58)
- `TemperatureSensor` removed pending further work (#93)
- `Buzzer.beep()` alias method added (#75)
- `Motor` PWM devices exposed, and `Robot` motor devices exposed (#107)

Release 0.9.0 (2015-10-25)

Fourth public beta

- Added source and values properties to all relevant classes (#76)
- Fix names of parameters in `Motor` constructor (#79)
- Added wrappers for LED groups on add-on boards (#81)

Release 0.8.0 (2015-10-16)

Third public beta

- Added generic `AnalogInputDevice` class along with specific classes for the `MCP3008` and `MCP3004` (#41)
- Fixed `DigitalOutputDevice.blink()` (#57)

Release 0.7.0 (2015-10-09)

Second public beta

Release 0.6.0 (2015-09-28)

First public beta

Release 0.5.0 (2015-09-24)

Release 0.4.0 (2015-09-23)

Release 0.3.0 (2015-09-22)

Release 0.2.0 (2015-09-21)

Initial release

License

Copyright 2015 Raspberry Pi Foundation.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- Neither the name of the copyright holder nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Index

Symbols

A

B

C

D

N

O

P

Q

R

S

T

U

V

W