

---

# **Gpiozero Documentation**

***Release 1.4.0***

**Ben Nuttall**

**Jul 26, 2017**



---

## Contents

---

<b>1</b>	<b>Installing GPIO Zero</b>	<b>1</b>
<b>2</b>	<b>Basic Recipes</b>	<b>3</b>
<b>3</b>	<b>Advanced Recipes</b>	<b>27</b>
<b>4</b>	<b>Configuring Remote GPIO</b>	<b>35</b>
<b>5</b>	<b>Remote GPIO Recipes</b>	<b>43</b>
<b>6</b>	<b>Source/Values</b>	<b>47</b>
<b>7</b>	<b>Command-line Tools</b>	<b>53</b>
<b>8</b>	<b>Frequently Asked Questions</b>	<b>61</b>
<b>9</b>	<b>Contributing</b>	<b>65</b>
<b>10</b>	<b>Development</b>	<b>67</b>
<b>11</b>	<b>API - Input Devices</b>	<b>69</b>
<b>12</b>	<b>API - Output Devices</b>	<b>81</b>
<b>13</b>	<b>API - SPI Devices</b>	<b>97</b>
<b>14</b>	<b>API - Boards and Accessories</b>	<b>105</b>
<b>15</b>	<b>API - Internal Devices</b>	<b>141</b>
<b>16</b>	<b>API - Generic Classes</b>	<b>145</b>
<b>17</b>	<b>API - Device Source Tools</b>	<b>151</b>
<b>18</b>	<b>API - Pi Information</b>	<b>159</b>
<b>19</b>	<b>API - Pins</b>	<b>163</b>
<b>20</b>	<b>API - Exceptions</b>	<b>177</b>
<b>21</b>	<b>Changelog</b>	<b>181</b>
<b>22</b>	<b>License</b>	<b>187</b>



# CHAPTER 1

---

## Installing GPIO Zero

---

GPIO Zero is installed by default in [Raspbian Jessie](#)<sup>1</sup> and [Raspbian x86](#)<sup>2</sup>, available from [raspberrypi.org](#)<sup>3</sup>. Follow these guides to installing on other operating systems, including for PCs using the *remote GPIO* (page 35) feature.

### Raspberry Pi

First, update your repositories list:

```
pi@raspberrypi:~$ sudo apt update
```

Then install the package for Python 3:

```
pi@raspberrypi:~$ sudo apt install python3-gpiozero
```

or Python 2:

```
pi@raspberrypi:~$ sudo apt install python-gpiozero
```

If you're using another operating system on your Raspberry Pi, you may need to use pip to install GPIO Zero instead. Install pip using [get-pip](#)<sup>4</sup> and then type:

```
pi@raspberrypi:~$ sudo pip3 install gpiozero
```

or for Python 2:

```
pi@raspberrypi:~$ sudo pip install gpiozero
```

To install GPIO Zero in a virtual environment, see the [Development](#) (page 67) page.

---

<sup>1</sup> <https://www.raspberrypi.org/downloads/raspbian/>

<sup>2</sup> <https://www.raspberrypi.org/blog/pixel-pc-mac/>

<sup>3</sup> <https://www.raspberrypi.org/downloads/>

<sup>4</sup> <https://pip.pypa.io/en/stable/installing/>

## PC/Mac

In order to use GPIO Zero's remote GPIO feature from a PC or Mac, you'll need to install GPIO Zero on that computer using pip. See the [\*Configuring Remote GPIO\*](#) (page 35) page for more information.

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

## Importing 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` (page 69) interface from GPIO Zero, it should be explicitly imported:

```
from gpiozero import Button
```

Now `Button` (page 69) 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)
```

## Pin Numbering

This library uses Broadcom (BCM) pin numbering for the GPIO pins, as opposed to physical (BOARD) numbering. Unlike in the `RPi.GPIO`<sup>5</sup> 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:

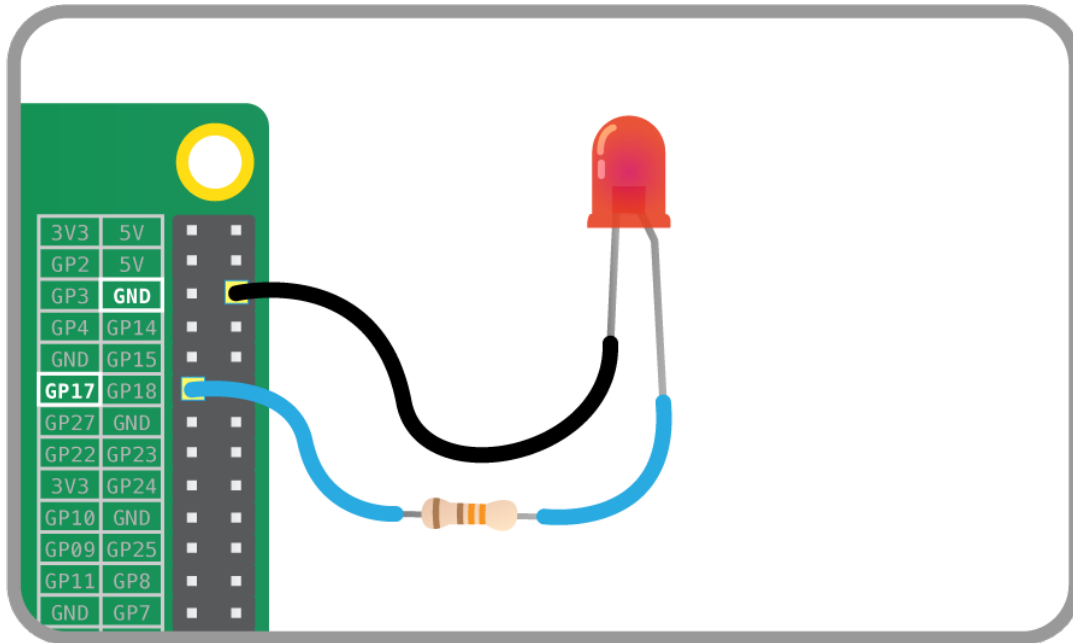
---

<sup>5</sup> <https://pypi.python.org/pypi/RPi.GPIO>





## LED



Turn an [LED](#) (page 81) 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()`<sup>6</sup>. See *How do I keep my script running?* (page 61) 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](#) (page 82) using values between 0 and 1:

<sup>6</sup> <https://docs.python.org/3.5/library/signal.html#signal.pause>

```

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

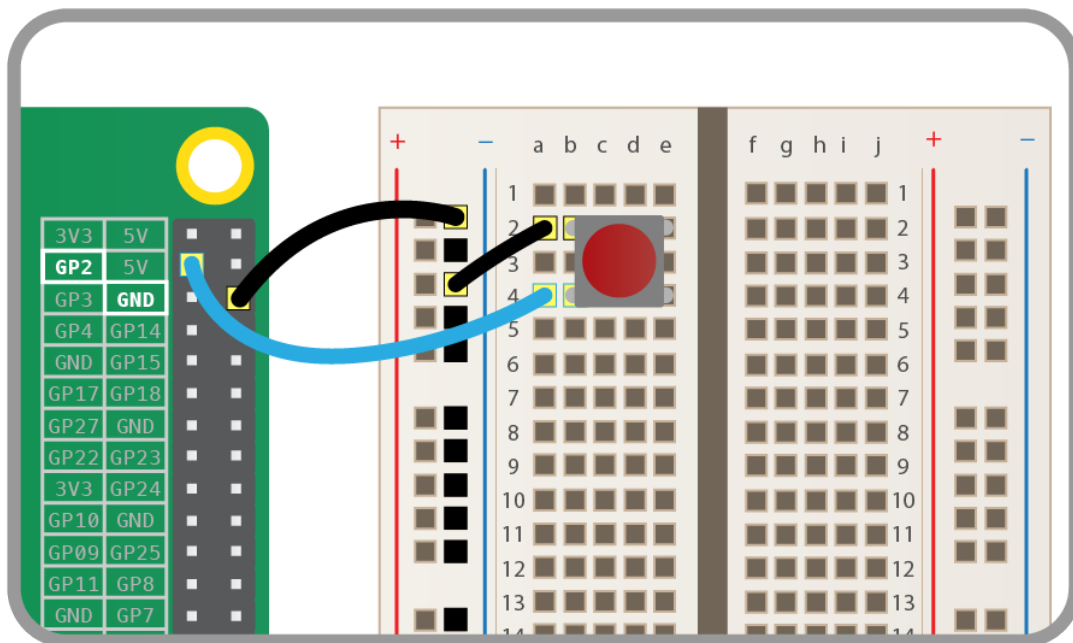
led = PWMLED(17)

led.pulse()

pause()

```

## Button



Check if a *Button* (page 69) 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:

```
from gpiozero import Button

button = Button(2)

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

Run a function every time the button is pressed:

```
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:

```
from gpiozero import Button
from signal import pause

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

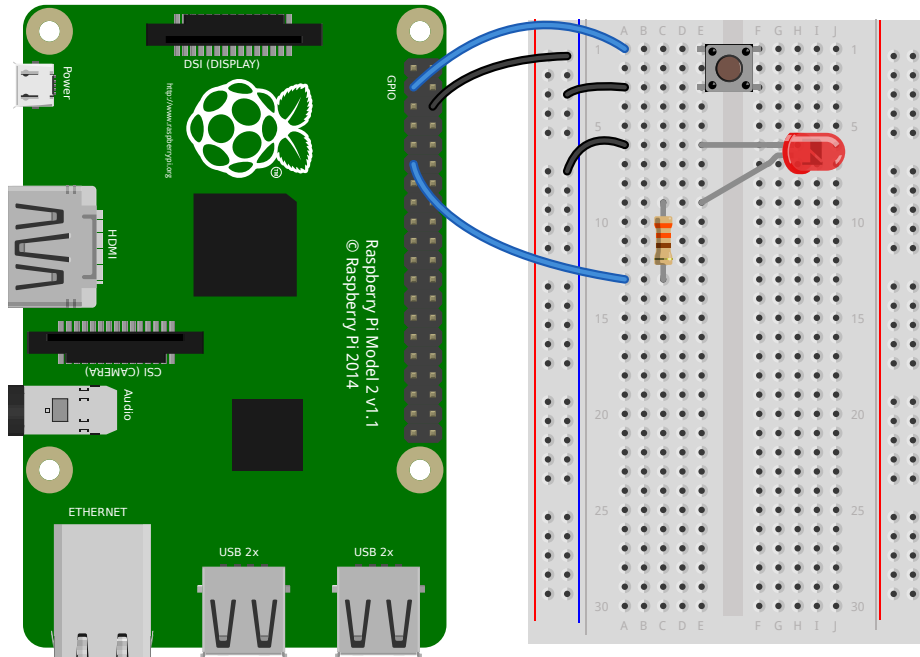
def say_goodbye():
    print("Goodbye!")

button = Button(2)

button.when_pressed = say_hello
button.when_released = say_goodbye

pause()
```

## Button controlled LED



Turn on an [LED](#) (page 81) when a [Button](#) (page 69) 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](#)<sup>7</sup> to take a picture using `button.when_pressed = camera.capture` would not work because the `capture()`<sup>8</sup> method requires an output parameter. However, this can be achieved using a custom function which requires no parameters:

<sup>7</sup> [https://picamera.readthedocs.io/en/latest/api\\_camera.html#picamera.PiCamera](https://picamera.readthedocs.io/en/latest/api_camera.html#picamera.PiCamera)

<sup>8</sup> [https://picamera.readthedocs.io/en/latest/api\\_camera.html#picamera.PiCamera.capture](https://picamera.readthedocs.io/en/latest/api_camera.html#picamera.PiCamera.capture)

```

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* (page 69) 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* (page 105):

```
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* (page 105) with `pwm=True` allows each LED's brightness to be controlled:

```
from gpiozero import LEDBoard
from signal import pause

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

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

pause()
```

See more *LEDBoard* (page 105) examples in the *advanced LEDBoard recipes* (page 27).

## LEDBarGraph

A collection of LEDs can be treated like a bar graph using *LEDBarGraph* (page 108):

```
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* (page 108) 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)

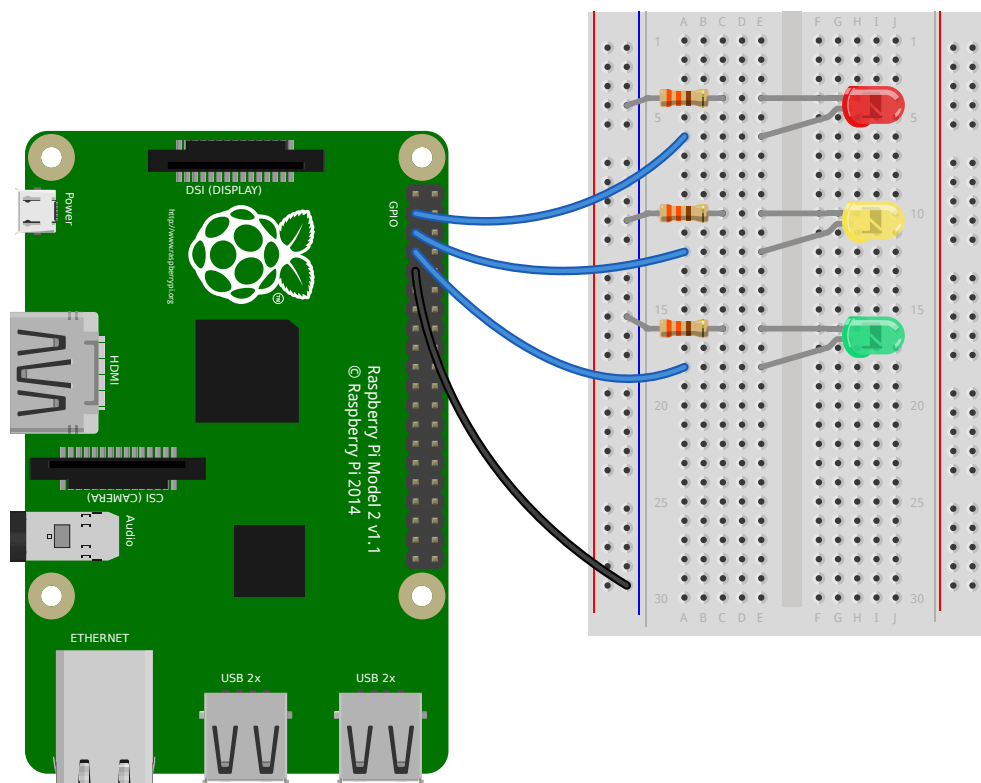
graph.value = 1/10 # (0.5, 0, 0, 0, 0)
```

```

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](#) (page 111) 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](#) (page 81) 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()
```

## 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
```



```

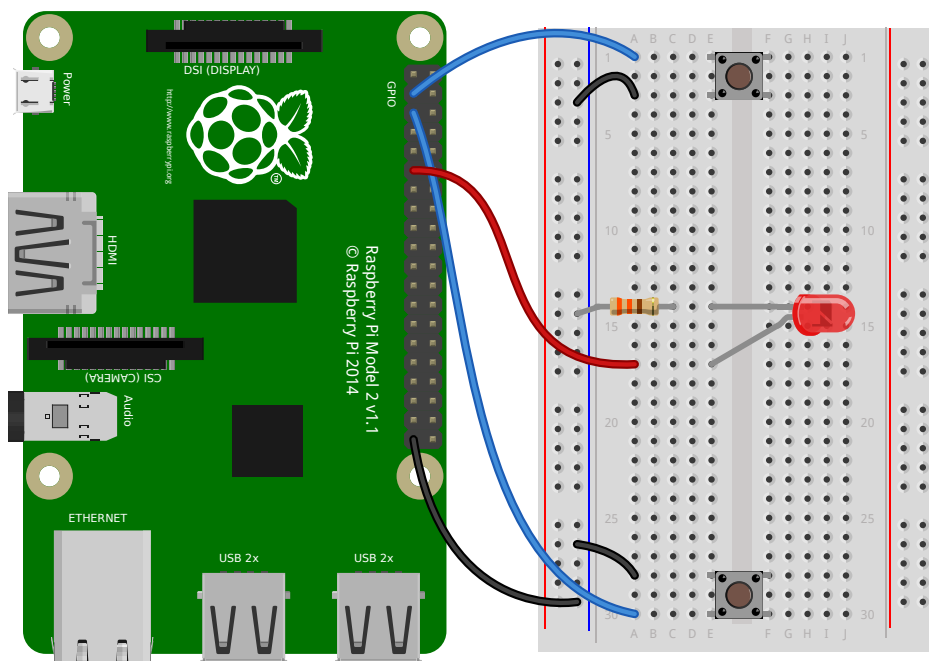
button = Button(2)
camera = PiCamera()

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](#)<sup>9</sup> 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:

```

<sup>9</sup> <https://www.raspberrypi.org/learning/quick-reaction-game/>

```
        print("Player 2 wins!")
        break

led.off()
```

See [Quick Reaction Game](#)<sup>10</sup> 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()

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

for button, sound in button_sounds.items():
    button.when_pressed = sound.play

pause()
```

See [GPIO Music Box](#)<sup>11</sup> for a full resource.

## All on when pressed

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

*FishDish* (page 125):

```
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* (page 126):

```
from gpiozero import TrafficHat
from signal import pause

th = TrafficHat()

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

---

<sup>10</sup> <https://www.raspberrypi.org/learning/quick-reaction-game/>

<sup>11</sup> <https://www.raspberrypi.org/learning/gpio-music-box/>

```
pause()
```

Using [LED](#) (page 81), [Buzzer](#) (page 86), and [Button](#) (page 69) 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]

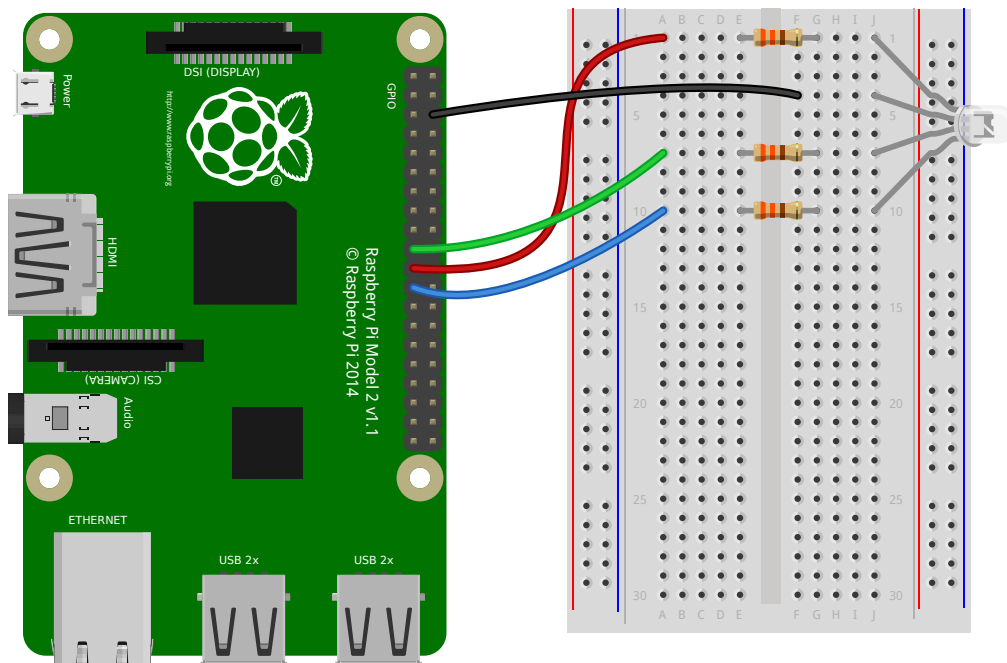
def things_on():
    for thing in things:
        thing.on()

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](#) (page 84):

```
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)

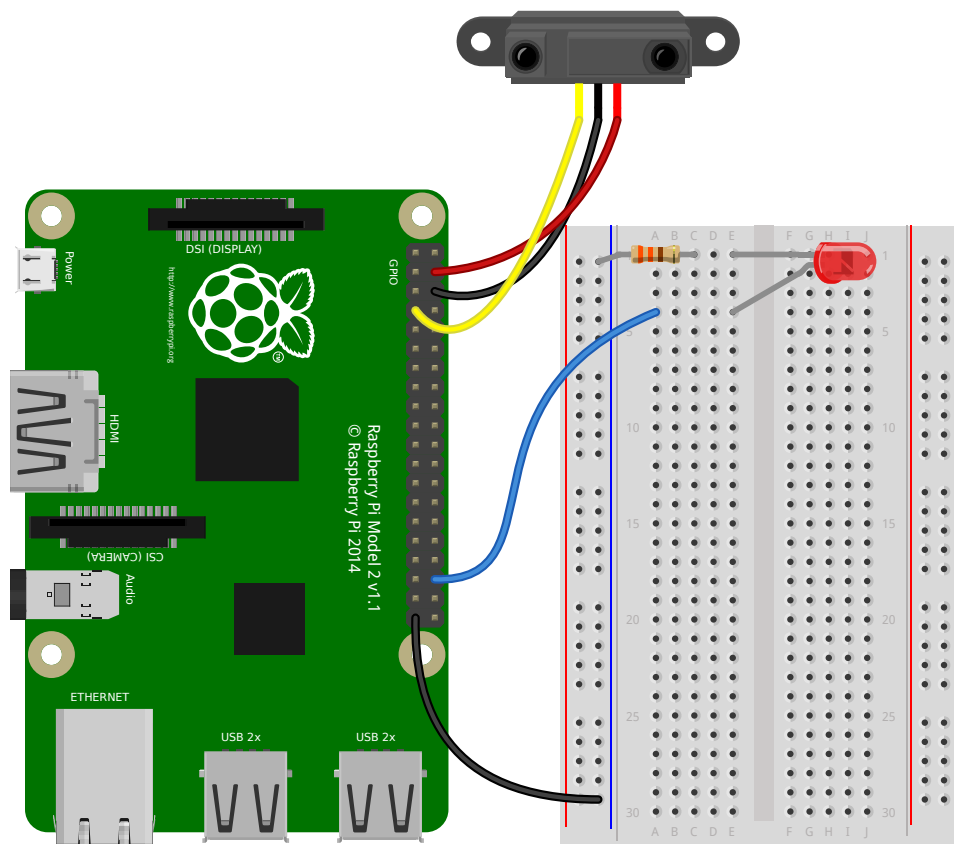
led.color = (0, 1, 0)  # full green
sleep(1)
led.color = (1, 0, 1)  # magenta
sleep(1)
led.color = (1, 1, 0)  # yellow
sleep(1)
led.color = (0, 1, 1)  # cyan
sleep(1)
led.color = (1, 1, 1)  # white
sleep(1)

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](#) (page 81) when a [MotionSensor](#) (page 72) detects motion:

```

from gpiozero import MotionSensor, LED
from signal import pause

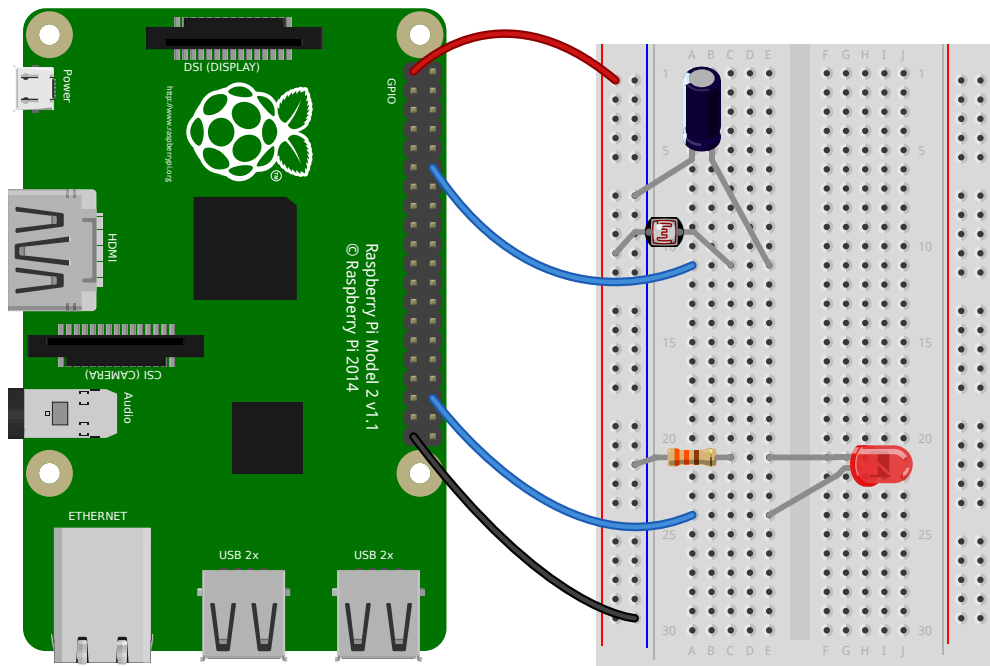
pir = MotionSensor(4)
led = LED(16)

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

pause()

```

## Light sensor



Have a *LightSensor* (page 74) 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* (page 82) 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* (page 75) 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:

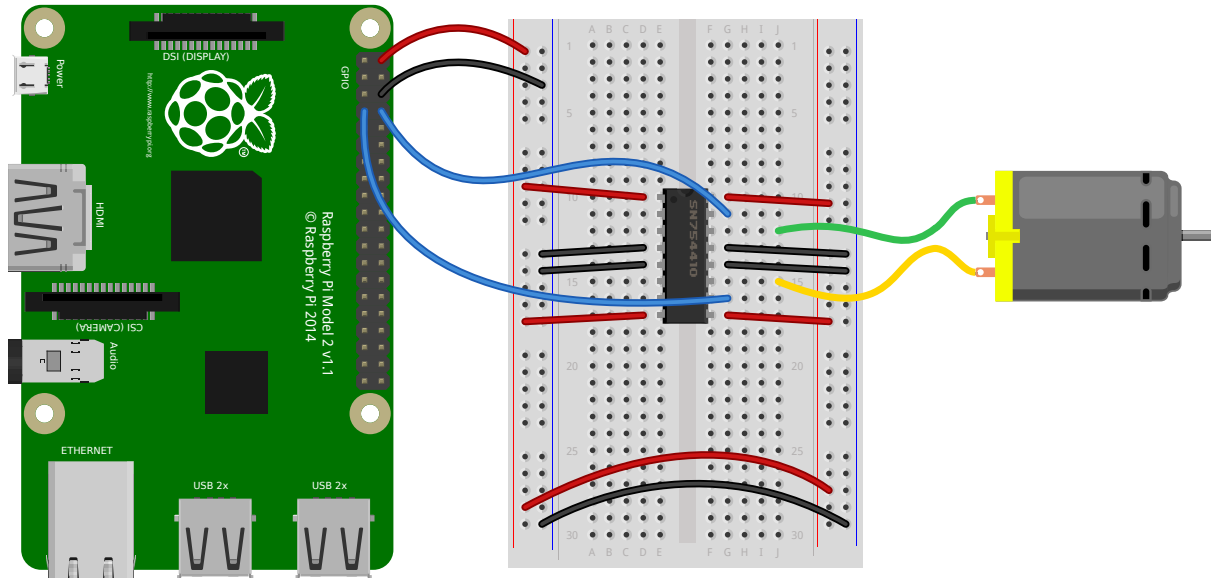
```
from gpiozero import DistanceSensor, LED
from signal import pause

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](#) (page 87) 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](#) (page 126) 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`<sup>12</sup> 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.

<sup>12</sup> <https://docs.python.org/3.5/library/curses.html#module-curses>

## 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:

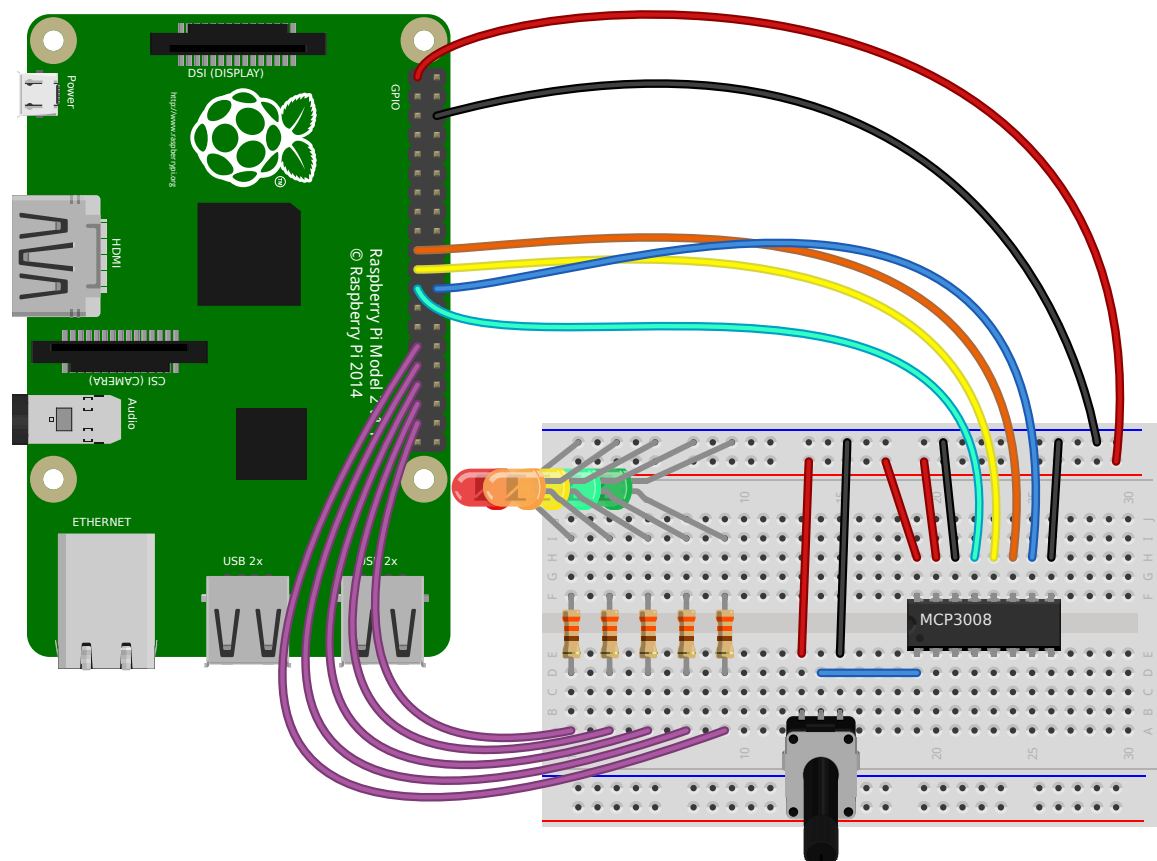
```
from gpiozero import Robot, MotionSensor
from signal import pause

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* (page 99) 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* (page 99) 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* (page 148) property rather than a *while*<sup>13</sup> loop:

<sup>13</sup> [https://docs.python.org/3.5/reference/compound\\_stmts.html#while](https://docs.python.org/3.5/reference/compound_stmts.html#while)

```
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()
```

---

**Note:** Please note the example above requires Python 3. In Python 2, `zip()`<sup>14</sup> doesn't support lazy evaluation so the script will simply hang.

---

## Timed heat lamp

If you have a pet (e.g. a tortoise) which requires a heat lamp to be switched on for a certain amount of time each day, you can use an [Energenie Pi-mote](#)<sup>15</sup> to remotely control the lamp, and the `TimeOfDay` (page 141) class to control the timing:

```
from gpiozero import Energenie, TimeOfDay
from datetime import time
from signal import pause

lamp = Energenie(1)
daytime = TimeOfDay(time(8), time(20))

lamp.source = daytime.values
lamp.source_delay = 60

pause()
```

## Internet connection status indicator

You can use a pair of green and red LEDs to indicate whether or not your internet connection is working. Simply use the `PingServer` (page 142) class to identify whether a ping to `google.com` is successful. If successful, the green LED is lit, and if not, the red LED is lit:

```
from gpiozero import LED, PingServer
from gpiozero.tools import negated
from signal import pause

green = LED(17)
red = LED(18)

google = PingServer('google.com')

green.source = google.values
green.source_delay = 60
red.source = negated(green.values)

pause()
```

---

<sup>14</sup> <https://docs.python.org/3.5/library/functions.html#zip>

<sup>15</sup> <https://energenie4u.co.uk/catalogue/product/ENER002-2PI>

---

## CPU Temperature Bar Graph

You can read the Raspberry Pi’s own CPU temperature using the built-in *CPUTemperature* (page 142) class, and display this on a “bar graph” of LEDs:

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

cpu = CPUTemperature(min_temp=50, max_temp=90)
leds = LEDBarGraph(2, 3, 4, 5, 6, 7, 8, pwm=True)

leds.source = cpu.values

pause()
```

## More recipes

Continue to:

- *Advanced Recipes* (page 27)
- *Remote GPIO Recipes* (page 43)



---

## Advanced Recipes

---

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

### LEDBoard

You can iterate over the LEDs in a *LEDBoard* (page 105) object one-by-one:

```
from gpiozero import LEDBoard
from time import sleep

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

for led in leds:
    led.on()
    sleep(1)
    led.off()
```

*LEDBoard* (page 105) also supports indexing. This means you can access the individual *LED* (page 81) objects using `leds[i]` where `i` is an integer from 0 up to (not including) the number of LEDs:

```
from gpiozero import LEDBoard
from time import sleep

leds = LEDBoard(2, 3, 4, 5, 6, 7, 8, 9)

leds[0].on() # first led on
sleep(1)
leds[7].on() # last led on
sleep(1)
leds[-1].off() # last led off
sleep(1)
```

This also means you can use slicing to access a subset of the LEDs:

```
from gpiozero import LEDBoard
from time import sleep
```

```
leds = LEDBoard(2, 3, 4, 5, 6, 7, 8, 9)

for led in leds[3:]: # leds 3 and onward
    led.on()
sleep(1)
leds.off()

for led in leds[:2]: # leds 0 and 1
    led.on()
sleep(1)
leds.off()

for led in leds[::2]: # even leds (0, 2, 4...)
    led.on()
sleep(1)
leds.off()

for led in leds[1::2]: # odd leds (1, 3, 5...)
    led.on()
sleep(1)
leds.off()
```

*LEDBoard* (page 105) objects can have their *LED* objects named upon construction. This means the individual LEDs can be accessed by their name:

```
from gpiozero import LEDBoard
from time import sleep

leds = LEDBoard(red=2, green=3, blue=4)

leds.red.on()
sleep(1)
leds.green.on()
sleep(1)
leds.blue.on()
sleep(1)
```

*LEDBoard* (page 105) objects can also be nested within other *LEDBoard* (page 105) objects:

```
from gpiozero import LEDBoard
from time import sleep

leds = LEDBoard(red=LEDBoard(top=2, bottom=3), green=LEDBoard(top=4, bottom=5))

leds.red.on() ## both reds on
sleep(1)
leds.green.on() # both greens on
sleep(1)
leds.off() # all off
sleep(1)
leds.red.top.on() # top red on
sleep(1)
leds.green.bottom.on() # bottom green on
sleep(1)
```

## Who's home indicator

Using a number of green-red LED pairs, you can show the status of who's home, according to which IP addresses you can ping successfully. Note that this assumes each person's mobile phone has a reserved IP address on the home router.



```

from gpiozero import PingServer, LEDBoard
from gpiozero.tools import negated
from signal import pause

status = LEDBoard(
    mum=LEDBoard(red=14, green=15),
    dad=LEDBoard(red=17, green=18),
    alice=LEDBoard(red=21, green=22)
)

statuses = {
    PingServer('192.168.1.5'): status.mum,
    PingServer('192.168.1.6'): status.dad,
    PingServer('192.168.1.7'): status.alice,
}

for server, leds in statuses.items():
    leds.green.source = server.values
    leds.green.source_delay = 60
    leds.red.source = negated(leds.green.values)

pause()

```

Alternatively, using the **STATUS Zero**<sup>16</sup> board:

```

from gpiozero import PingServer, StatusZero
from gpiozero.tools import negated
from signal import pause

status = StatusZero('mum', 'dad', 'alice')

statuses = {
    PingServer('192.168.1.5'): status.mum,
    PingServer('192.168.1.6'): status.dad,
    PingServer('192.168.1.7'): status.alice,
}

for server, leds in statuses.items():
    leds.green.source = server.values
    leds.green.source_delay = 60
    leds.red.source = negated(leds.green.values)

pause()

```

## 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)

```

<sup>16</sup> <https://thepihut.com/status>

```
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`<sup>17</sup>. Install with `sudo pip3 install travispy`.

## Button controlled robot

Alternatively to the examples in the simple recipes, you can use four buttons to program the directions and add a fifth button to process them in turn, like a Bee-Bot or Turtle robot.

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

robot = Robot((17, 18), (22, 23))

left = Button(2)
right = Button(3)
forward = Button(4)
backward = Button(5)
go = Button(6)

instructions = []

def add_instruction(btn):
    instructions.append({
        left: (-1, 1),
        right: (1, -1),
        forward: (1, 1),
        backward: (-1, -1),
    }[btn])

def do_instructions():
    instructions.append((0, 0))
    robot.source_delay = 0.5
    robot.source = instructions
    sleep(robot.source_delay * len(instructions))
    del instructions[:]

go.when_pressed = do_instructions
for button in (left, right, forward, backward):
    button.when_pressed = add_instruction

pause()
```

## Robot controlled by 2 potentiometers

Use two potentiometers to control the left and right motor speed of a robot:

---

<sup>17</sup> <https://travispy.readthedocs.io/>

```

from gpiozero import Robot, MCP3008
from signal import pause

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

left = MCP3008(0)
right = MCP3008(1)

robot.source = zip(left.values, right.values)

pause()

```

**Note:** Please note the example above requires Python 3. In Python 2, `zip()`<sup>18</sup> doesn't support lazy evaluation so the script will simply hang.

To include reverse direction, scale the potentiometer values from 0-1 to -1-1:

```

from gpiozero import Robot, MCP3008
from gpiozero.tools import scaled
from signal import pause

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

left = MCP3008(0)
right = MCP3008(1)

robot.source = zip(scaled(left.values, -1, 1), scaled(right.values, -1, 1))

pause()

```

## BlueDot LED

BlueDot is a Python library an Android app which allows you to easily add Bluetooth control to your Raspberry Pi project. A simple example to control a LED using the BlueDot app:

```

from blue dot import BlueDot
from gpiozero import LED

bd = BlueDot()
led = LED(17)

while True:
    bd.wait_for_press()
    led.on()
    bd.wait_for_release()
    led.off()

```

Note this recipe requires `blue dot` and the associated Android app. See the [BlueDot documentation](https://blue dot.readthedocs.io/en/latest/index.html)<sup>19</sup> for installation instructions.

<sup>18</sup> <https://docs.python.org/3.5/library/functions.html#zip>

<sup>19</sup> <https://blue dot.readthedocs.io/en/latest/index.html>

## BlueDot robot

You can create a Bluetooth controlled robot which moves forward when the dot is pressed and stops when it is released:

```
from bluebot import BlueDot
from gpiozero import Robot
from signal import pause

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

def move(pos):
    if pos.top:
        robot.forward(pos.distance)
    elif pos.bottom:
        robot.backward(pos.distance)
    elif pos.left:
        robot.left(pos.distance)
    elif pos.right:
        robot.right(pos.distance)

bd.when_pressed = move
bd.when_moved = move
bd.when_released = robot.stop

pause()
```

Or a more advanced example including controlling the robot's speed and precise direction:

```
from gpiozero import Robot
from bluebot import BlueDot
from signal import pause

def pos_to_values(x, y):
    left = y if x > 0 else y + x
    right = y if x < 0 else y - x
    return (clamped(left), clamped(right))

def clamped(v):
    return max(-1, min(1, v))

def drive():
    while True:
        if bd.is_pressed:
            x, y = bd.position.x, bd.position.y
            yield pos_to_values(x, y)
        else:
            yield (0, 0)

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

robot.source = drive()

pause()
```

## 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).

---



---

## Configuring Remote GPIO

---

GPIO Zero supports a number of different pin implementations (low-level pin libraries which deal with the GPIO pins directly). By default, the [RPi.GPIO](#)<sup>20</sup> library is used (assuming it is installed on your system), but you can optionally specify one to use. For more information, see the [API - Pins](#) (page 163) documentation page.

One of the pin libraries supported, [pigpio](#)<sup>21</sup>, provides the ability to control GPIO pins remotely over the network, which means you can use GPIO Zero to control devices connected to a Raspberry Pi on the network. You can do this from another Raspberry Pi, or even from a PC.

See the [Remote GPIO Recipes](#) (page 43) page for examples on how remote pins can be used.

## Preparing the Raspberry Pi

If you're using Raspbian (desktop - not Raspbian Lite) then you have everything you need to use the remote GPIO feature. If you're using Raspbian Lite, or another distribution, you'll need to install pigpio:

```
$ sudo apt install pigpio
```

Alternatively, pigpio is available from [abyz.co.uk](#)<sup>22</sup>.

You'll need to launch the pigpio daemon on the Raspberry Pi to allow remote connections. You can do this in three different ways. Most users will find the desktop method the easiest (and can skip to the next section).

## Desktop

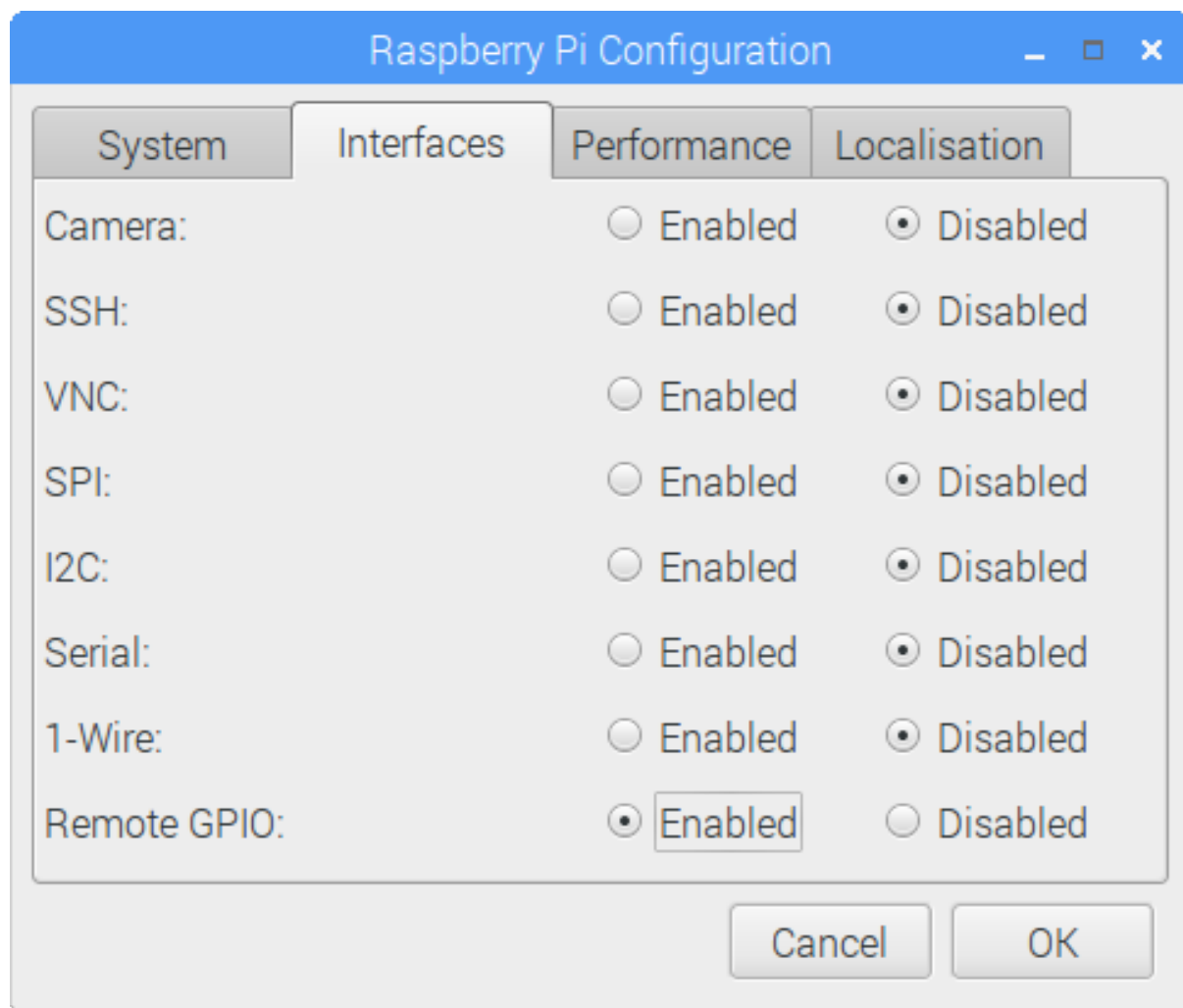
On the Raspbian desktop image, enable **Remote GPIO** in the Raspberry Pi configuration tool:

---

<sup>20</sup> <https://pypi.python.org/pypi/RPi.GPIO>

<sup>21</sup> <http://abyz.co.uk/rpi/pigpio/python.html>

<sup>22</sup> <http://abyz.co.uk/rpi/pigpio/download.html>



This will launch the pigpio daemon automatically.

### Command-line: raspi-config

Alternatively, enter `sudo raspi-config` on the command line, and enable Remote GPIO. This will also launch the pigpio daemon automatically.

### Command-line: manual

Another option is to launch the pigpio daemon manually:

```
$ sudo pigpiod
```

This is for single-use and will not persist after a reboot. However, this method can be used to allow connections from a specific IP address, using the `-n` flag. For example:

```
$ sudo pigpiod -n localhost # allow localhost only
$ sudo pigpiod -n 192.168.1.65 # allow 192.168.1.65 only
$ sudo pigpiod -n localhost -n 192.168.1.65 # allow localhost and 192.168.1.65 only
```

To automate running the daemon at boot time, run:

```
$ sudo systemctl enable pigpiod
```



## Preparing the control computer

If the control computer (the computer you're running your Python code from) is a Raspberry Pi running Raspbian (or a PC running Raspbian x86), then you have everything you need. If you're using another Linux distribution, Mac OS or Windows then you'll need to install the `pigpio` Python library on the PC.

### Raspberry Pi

First, update your repositories list:

```
$ sudo apt update
```

Then install GPIO Zero and the `pigpio` library for Python 3:

```
$ sudo apt install python3-gpiozero python3-pigpio
```

or Python 2:

```
$ sudo apt install python-gpiozero python-pigpio
```

Alternatively, install with `pip`:

```
$ sudo pip3 install gpiozero pigpio
```

or for Python 2:

```
$ sudo pip install gpiozero pigpio
```

### Linux

First, update your distribution's repositories list. For example:

```
$ sudo apt update
```

Then install `pip` for Python 3:

```
$ sudo apt install python3-pip
```

or Python 2:

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

(Alternatively, install `pip` with [get-pip](https://pip.pypa.io/en/stable/installing/)<sup>23</sup>.)

Next, install GPIO Zero and `pigpio` for Python 3:

```
$ sudo pip3 install gpiozero pigpio
```

or Python 2:

```
$ sudo pip install gpiozero pigpio
```

---

<sup>23</sup> <https://pip.pypa.io/en/stable/installing/>

## Mac OS

First, install pip. If you installed Python 3 using brew, you will already have pip. If not, install pip with [get-pip](#)<sup>24</sup>. Next, install GPIO Zero and pigpio with pip:

```
$ pip3 install gpiozero pigpio
```

Or for Python 2:

```
$ pip install gpiozero pigpio
```

## Windows

First, install pip by [following this guide](#)<sup>25</sup>. Next, install GPIO Zero and pigpio with pip:

```
C:\Users\user1> pip install gpiozero pigpio
```

## Environment variables

The simplest way to use devices with remote pins is to set the `PIGPIO_ADDR` environment variable to the IP address of the desired Raspberry Pi. You must run your Python script or launch your development environment with the environment variable set using the command line. For example, one of the following:

```
$ PIGPIO_ADDR=192.168.1.3 python3 hello.py
$ PIGPIO_ADDR=192.168.1.3 python3
$ PIGPIO_ADDR=192.168.1.3 ipython3
$ PIGPIO_ADDR=192.168.1.3 idle3 &
```

If you are running this from a PC (not a Raspberry Pi) with gpiozero and the pigpio Python library installed, this will work with no further configuration. However, if you are running this from a Raspberry Pi, you will also need to ensure the default pin factory is set to `PiGPIOFactory`. If `RPi.GPIO` is installed, this will be selected as the default pin factory, so either uninstall it, or use another environment variable to set it to `PiGPIOFactory`:

```
$ GPIOZERO_PIN_FACTORY=pigpio PIGPIO_ADDR=192.168.1.3 python3 hello.py
```

This usage will set the pin factory to `PiGPIOFactory` with a default host of `192.168.1.3`. The pin factory can be changed inline in the code, as seen in the following sections.

With this usage, you can write gpiozero code like you would on a Raspberry Pi, with no modifications needed. For example:

```
from gpiozero import LED
from time import sleep

red = LED(17)

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

When run with:

---

<sup>24</sup> <https://pip.pypa.io/en/stable/installing/>

<sup>25</sup> <https://www.raspberrypi.org/learning/using-pip-on-windows/worksheet/>

```
$ PIGPIO_ADDR=192.168.1.3 python3 led.py
```

will flash the LED connected to pin 17 of the Raspberry Pi with the IP address 192.168.1.3. And:

```
$ PIGPIO_ADDR=192.168.1.4 python3 led.py
```

will flash the LED connected to pin 17 of the Raspberry Pi with the IP address 192.168.1.4, without any code changes, as long as the Raspberry Pi has the pigpio daemon running.

**Note:** When running code directly on a Raspberry Pi, any pin factory can be used (assuming the relevant library is installed), but when a device is used remotely, only `PiGPIOFactory` can be used, as pigpio is the only pin library which supports remote GPIO.

## Pin objects

An alternative (or additional) method of configuring gpiozero objects to use remote pins is to create instances of `PiGPIOFactory` objects, and use them when instantiating device objects. For example, with no environment variables set:

```
from gpiozero import LED
from gpiozero.pins.pigpio import PiGPIOFactory
from time import sleep

factory = PiGPIOFactory(host='192.168.1.3')
led = LED(17, pin_factory=factory)

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

This allows devices on multiple Raspberry Pis to be used in the same script:

```
from gpiozero import LED
from gpiozero.pins.pigpio import PiGPIOFactory
from time import sleep

factory3 = PiGPIOFactory(host='192.168.1.3')
factory4 = PiGPIOFactory(host='192.168.1.4')
led_1 = LED(17, pin_factory=factory3)
led_2 = LED(17, pin_factory=factory4)

while True:
    led_1.on()
    led_2.off()
    sleep(1)
    led_1.off()
    led_2.on()
    sleep(1)
```

You can, of course, continue to create gpiozero device objects as normal, and create others using remote pins. For example, if run on a Raspberry Pi, the following script will flash an LED on the controller Pi, and also on another Pi on the network:

```
from gpiozero import LED
from gpiozero.pins.pigpio import PiGPIOFactory
from time import sleep
```

```
remote_factory = PiGPIOFactory(host='192.168.1.3')
led_1 = LED(17) # local pin
led_2 = LED(17, pin_factory=remote_factory) # remote pin

while True:
    led_1.on()
    led_2.off()
    sleep(1)
    led_1.off()
    led_2.on()
    sleep(1)
```

Alternatively, when run with the environment variables `GPIOZERO_PIN_FACTORY=pigpio` and `PIGPIO_ADDR=192.168.1.3` set, the following script will behave exactly the same as the previous one:

```
from gpiozero import LED
from gpiozero.pins.pigpio import RPiGPIOFactory
from time import sleep

local_factory = RPiGPIOFactory()
led_1 = LED(17, pin_factory=local_factory) # local pin
led_2 = LED(17) # remote pin

while True:
    led_1.on()
    led_2.off()
    sleep(1)
    led_1.off()
    led_2.on()
    sleep(1)
```

Of course, multiple IP addresses can be used:

```
from gpiozero import LED
from gpiozero.pins.pigpio import PiGPIOFactory
from time import sleep

factory3 = PiGPIOFactory(host='192.168.1.3')
factory4 = PiGPIOFactory(host='192.168.1.4')

led_1 = LED(17) # local pin
led_2 = LED(17, pin_factory=factory3) # remote pin on one pi
led_3 = LED(17, pin_factory=factory4) # remote pin on another pi

while True:
    led_1.on()
    led_2.off()
    led_3.on()
    sleep(1)
    led_1.off()
    led_2.on()
    led_3.off()
    sleep(1)
```

Note that these examples use the `LED` (page 81) class, which takes a `pin` argument to initialise. Some classes, particularly those representing HATs and other add-on boards, do not require their pin numbers to be specified. However, it is still possible to use remote pins with these devices, either using environment variables, `Device.pin_factory`, or the `pin_factory` keyword argument:

```
import gpiozero
from gpiozero import TrafficHat
from gpiozero.pins.pigpio import PiGPIOFactory
from time import sleep

gpiozero.Device.pin_factory = PiGPIOFactory(host='192.168.1.3')
th = TrafficHat() # traffic hat on 192.168.1.3 using remote pins
```

This also allows you to swap between two IP addresses and create instances of multiple HATs connected to different Pis:

```
import gpiozero
from gpiozero import TrafficHat
from gpiozero.pins.pigpio import PiGPIOFactory
from time import sleep

remote_factory = PiGPIOFactory(host='192.168.1.3')

th_1 = TrafficHat() # traffic hat using local pins
th_2 = TrafficHat(pin_factory=remote_factory) # traffic hat on 192.168.1.3 using
↪remote pins
```

You could even use a HAT which is not supported by GPIO Zero (such as the [Sense HAT](#)<sup>26</sup>) on one Pi, and use remote pins to control another over the network:

```
from gpiozero import MotionSensor
from gpiozero.pins.pigpio import PiGPIOFactory
from sense_hat import SenseHat

remote_factory = PiGPIOFactory(host='192.198.1.4')
pir = MotionSensor(4, pin_factory=remote_factory) # remote motion sensor
sense = SenseHat() # local sense hat

while True:
    pir.wait_for_motion()
    sense.show_message(sense.temperature)
```

Note that in this case, the Sense HAT code must be run locally, and the GPIO remotely.

## Pi Zero USB OTG

The [Raspberry Pi Zero](#)<sup>27</sup> and [Pi Zero W](#)<sup>28</sup> feature a USB OTG port, allowing users to configure the device as (amongst other things) an Ethernet device. In this mode, it is possible to control the Pi Zero's GPIO pins over USB from another computer using remote pins.

First, configure the boot partition of the SD card:

1. Edit `config.txt` and add `dtoverlay=dwc2` on a new line, then save the file.
2. Create an empty file called `ssh` (no file extension) and save it in the boot partition.
3. Edit `cmdline.txt` and insert `modules-load=dwc2,g_ether` after `rootwait`.

(See guides on [blog.gbaman.info](#)<sup>29</sup> and [learn.adafruit.com](#)<sup>30</sup> for more detailed instructions)

Then connect the Pi Zero to your computer using a micro USB cable (connecting it to the USB port, not the power port). You'll see the indicator LED flashing as the Pi Zero boots. When it's ready, you will be able to ping and

<sup>26</sup> <https://www.raspberrypi.org/products/sense-hat/>

<sup>27</sup> <https://www.raspberrypi.org/products/raspberry-pi-zero/>

<sup>28</sup> <https://www.raspberrypi.org/products/raspberry-pi-zero-w/>

<sup>29</sup> <http://blog.gbaman.info/?p=791>

<sup>30</sup> <https://learn.adafruit.com/turning-your-raspberry-pi-zero-into-a-usb-gadget/ethernet-gadget>

SSH into it using the hostname `raspberrypi.local`. SSH into the Pi Zero, install `pigpio` and run the `pigpio` daemon.

Then, drop out of the SSH session and you can run Python code on your computer to control devices attached to the Pi Zero, referencing it by its hostname (or IP address if you know it), for example:

```
$ GPIOZERO_PIN_FACTORY=pigpio PIGPIO_ADDR=raspberrypi.local python3 led.py
```

---

## Remote GPIO Recipes

---

The following recipes demonstrate some of the capabilities of the remote GPIO feature of the GPIO Zero library. Before you start following these examples, please read up on preparing your Pi and your host PC to work with *Configuring Remote GPIO* (page 35).

Please note that all recipes are written assuming Python 3. Recipes *may* work under Python 2, but no guarantees!

### LED + Button

Let a button on one Raspberry Pi control the LED of another:

```
from gpiozero import LED
from gpiozero.pins.pigpio import PiGPIOFactory
from signal import pause

factory = PiGPIOFactory(host='192.168.1.3')

button = Button(2)
led = LED(17, pin_factory=factory)

led.source = button.values

pause()
```

### LED + 2 Buttons

The LED will come on when both buttons are pressed:

```
from gpiozero import LED
from gpiozero.pins.pigpio import PiGPIOFactory
from gpiozero.tools import all_values
from signal import pause

factory3 = PiGPIOFactory(host='192.168.1.3')
factory4 = PiGPIOFactory(host='192.168.1.4')
```

```
led = LED(17)
button_1 = Button(17, pin_factory=factory3)
button_2 = Button(17, pin_factory=factory4)

led.source = all_values(button_1.values, button_2.values)

pause()
```

## Multi-room motion alert

Install a Raspberry Pi with a motion sensor in each room of your house, and have an LED indicator showing when there's motion in each room:

```
from gpiozero import LEDBoard, MotionSensor
from gpiozero.pins.pigpio import PiGPIOFactory
from signal import pause

ips = ['192.168.1.3', '192.168.1.4', '192.168.1.5', '192.168.1.6']
remotes = [PiGPIOFactory(host=ip) for ip in ips]

leds = LEDBoard(2, 3, 4, 5) # leds on this pi
sensors = [MotionSensor(17, pin_factory=r) for r in remotes] # remote sensors

for led, sensor in zip(leds, sensors):
    led.source = sensor.values

pause()
```

## Multi-room doorbell

Install a Raspberry Pi with a buzzer attached in each room you want to hear the doorbell, and use a push button as the doorbell:

```
from gpiozero import LEDBoard, MotionSensor
from gpiozero.pins.pigpio import PiGPIOFactory
from signal import pause

ips = ['192.168.1.3', '192.168.1.4', '192.168.1.5', '192.168.1.6']
remotes = [PiGPIOFactory(host=ip) for ip in ips]

button = Button(17) # button on this pi
buzzers = [Buzzer(pin, pin_factory=r) for r in remotes] # buzzers on remote pins

for buzzer in buzzers:
    buzzer.source = button.values

pause()
```

This could also be used as an internal doorbell (tell people it's time for dinner from the kitchen).

## Remote button robot

Similarly to the simple recipe for the button controlled robot, this example uses four buttons to control the direction of a robot. However, using remote pins for the robot means the control buttons can be separate from the robot:



```

from gpiozero import Button, Robot
from gpiozero.pins.pigpio import PiGPIOFactory
from signal import pause

factory = PiGPIOFactory(host='192.168.1.17')
robot = Robot(left=(4, 14), right=(17, 18), pin_factory=factory) # remote pins

# local buttons
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()

```

## Light sensor + Sense HAT

The [Sense HAT<sup>31</sup>](https://www.raspberrypi.org/products/sense-hat/) (not supported by GPIO Zero) includes temperature, humidity and pressure sensors, but no light sensor. Remote GPIO allows an external light sensor to be used as well. The Sense HAT LED display can be used to show different colours according to the light levels:

```

from gpiozero import LightSensor
from gpiozero.pins.pigpio import PiGPIOFactory
from sense_hat import SenseHat

remote_factory = PiGPIOFactory(host='192.168.1.4')
light = LightSensor(4, pin_factory=remote_factory) # remote motion sensor
sense = SenseHat() # local sense hat

blue = (0, 0, 255)
yellow = (255, 255, 0)

while True:
    if light.value > 0.5:
        sense.clear(yellow)
    else:
        sense.clear(blue)

```

Note that in this case, the Sense HAT code must be run locally, and the GPIO remotely.

<sup>31</sup> <https://www.raspberrypi.org/products/sense-hat/>



## CHAPTER 6

---

### Source/Values

---

GPIO Zero provides a method of using the declarative programming paradigm to connect devices together: feeding the values of one device into another, for example the values of a button into an LED:

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

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

led.source = button.values

pause()
```

which is equivalent to:

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

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

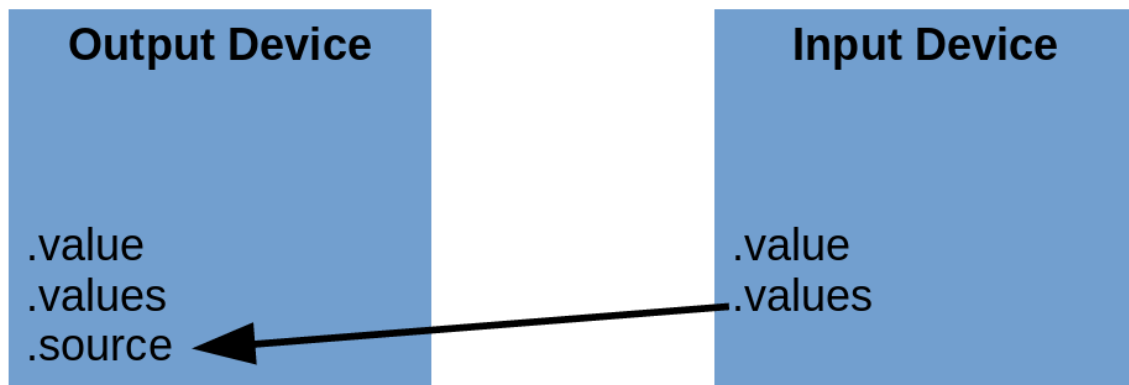
while True:
    led.value = button.value
    sleep(0.01)
```

Every device has a *value* (page 147) property (the device's current value). Input devices can only have their values read, but output devices can also have their value set to alter the state of the device:

```
>>> led = PWMLED(17)
>>> led.value # LED is initially off
0.0
>>> led.on() # LED is now on
>>> led.value
1.0
>>> led.value = 0 # LED is now off
```

Every device also has a *values* (page 148) property (a generator continuously yielding the device's current value). All output devices have a *source* (page 148) property which can be set to any iterator. The device will iterate over the values provided, setting the device's value to each element at a rate specified in the *source\_delay*

(page 148) property.



The most common use case for this is to set the source of an output device to the values of an input device, like the example above. A more interesting example would be a potentiometer controlling the brightness of an LED:

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

led = PWMLED(17)
pot = MCP3008()

led.source = pot.values

pause()
```

It is also possible to set an output device's `source` (page 148) to the `values` (page 148) of another output device, to keep them matching:

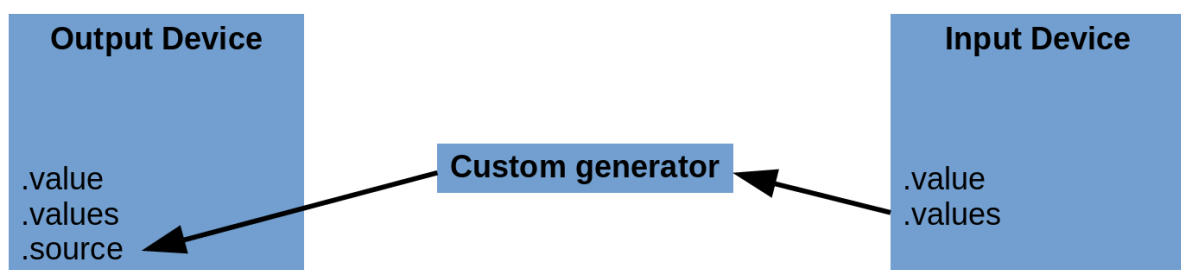
```
from gpiozero import LED, Button
from signal import pause

red = LED(14)
green = LED(15)
button = Button(17)

red.source = button.values
green.source = red.values

pause()
```

The device's values can also be processed before they are passed to the `source`:



For example:

```

from gpiozero import Button, LED
from signal import pause

def opposite(values):
    for value in values:
        yield not value

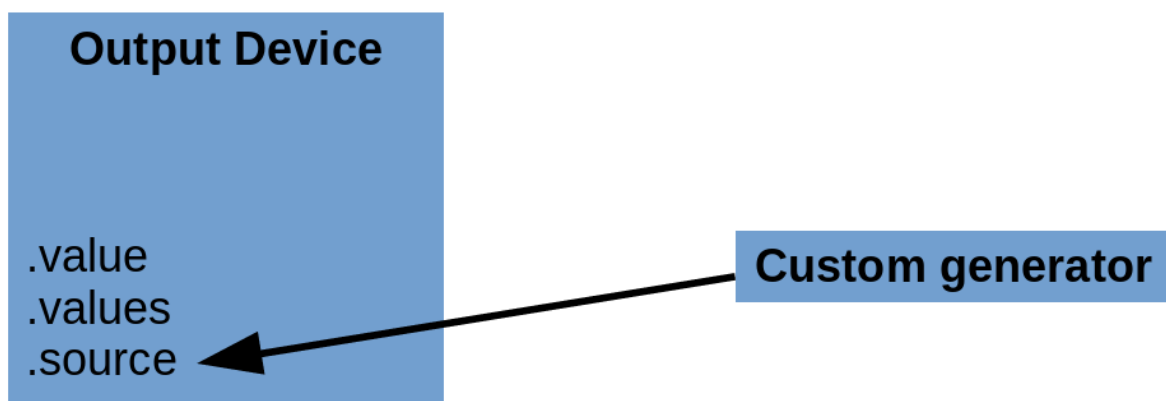
led = LED(4)
btn = Button(17)

led.source = opposite(btn.values)

pause()

```

Alternatively, a custom generator can be used to provide values from an artificial source:



For example:

```

from gpiozero import LED
from random import randint
from signal import pause

def rand():
    while True:
        yield randint(0, 1)

led = LED(17)
led.source = rand()

pause()

```

If the iterator is infinite (i.e. an infinite generator), the elements will be processed until the `source` (page 148) is changed or set to `None`.

If the iterator is finite (e.g. a list), this will terminate once all elements are processed (leaving the device's value at the final element):

```

from gpiozero import LED
from signal import pause

led = LED(17)
led.source = [1, 0, 1, 1, 1, 0, 0, 1, 0, 1]

pause()

```

## Composite devices

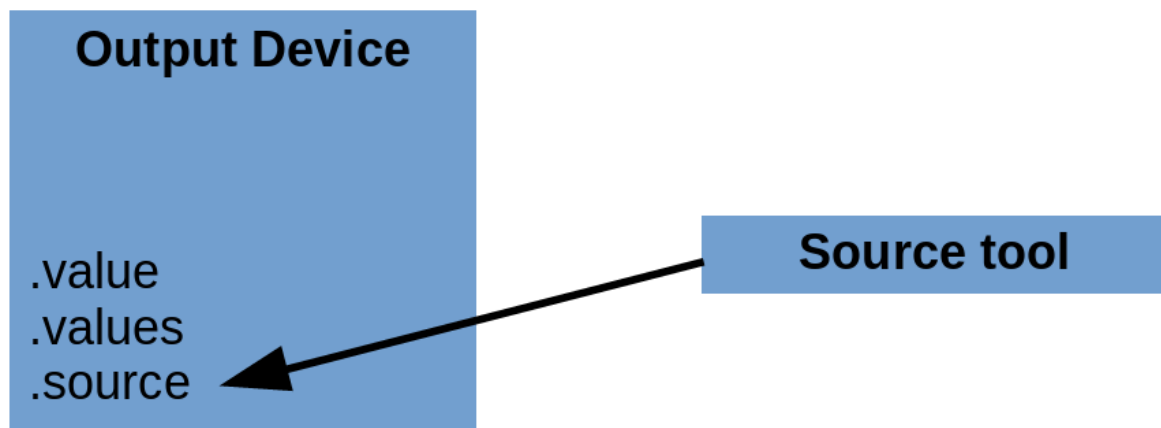
Most devices have a *value* (page 147) range between 0 and 1. Some have a range between -1 and 1 (e.g. *Motor* (page 87)). The *value* (page 147) of a composite device is a namedtuple of such values. For example, the *Robot* (page 126) class:

```
>>> from gpiozero import Robot
>>> robot = Robot(left=(14, 15), right=(17, 18))
>>> robot.value
RobotValue(left_motor=0.0, right_motor=0.0)
>>> tuple(robot.value)
(0.0, 0.0)
>>> robot.forward()
>>> tuple(robot.value)
(1.0, 1.0)
>>> robot.backward()
>>> tuple(robot.value)
(-1.0, -1.0)
>>> robot.value = (1, 1) # robot is now driven forwards
```

## Source Tools

GPIO Zero provides a set of ready-made functions for dealing with source/values, called source tools. These are available by importing from *gpiozero.tools* (page 151).

Some of these source tools are artificial sources which require no input:



In this example, random values between 0 and 1 are passed to the LED, giving it a flickering candle effect:

```
from gpiozero import PWMLED
from gpiozero.tools import random_values
from signal import pause

led = PWMLED(4)
led.source = random_values()
led.source_delay = 0.1

pause()
```

Some tools take a single source and process its values:



In this example, the LED is lit only when the button is not pressed:

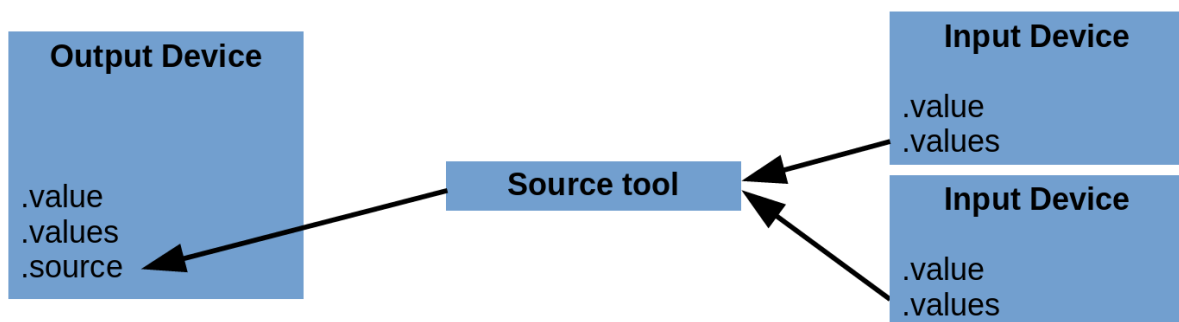
```
from gpiozero import Button, LED
from gpiozero.tools import negated
from signal import pause

led = LED(4)
btn = Button(17)

led.source = negated(btn.values)

pause()
```

Some tools combine the values of multiple sources:



In this example, the LED is lit only if both buttons are pressed (like an [AND](https://en.wikipedia.org/wiki/AND_gate)<sup>32</sup> gate):

```
from gpiozero import Button, LED
from gpiozero.tools import all_values
from signal import pause

button_a = Button(2)
button_b = Button(3)
led = LED(17)

led.source = all_values(button_a.values, button_b.values)

pause()
```

<sup>32</sup> [https://en.wikipedia.org/wiki/AND\\_gate](https://en.wikipedia.org/wiki/AND_gate)





## CHAPTER 7

---

### Command-line Tools

---

The `gpiozero` package contains a database of information about the various revisions of Raspberry Pi. This is queried by the `pinout` command-line tool to output details of the GPIO pins available.

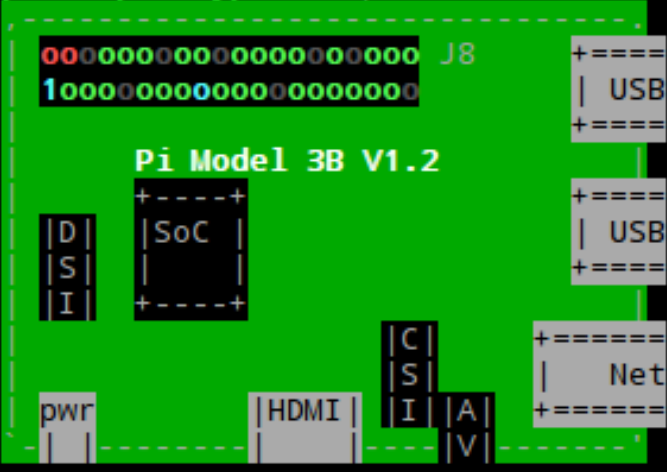


## pinout

```

pi@raspberrypi: ~
File Edit Tabs Help
pi@raspberrypi:~ $ pinout

```



```

Revision          : a02082
SoC               : BCM2837
RAM              : 1024Mb
Storage          : MicroSD
USB ports        : 4 (excluding power)
Ethernet ports   : 1
Wi-fi            : True
Bluetooth        : True
Camera ports (CSI) : 1
Display ports (DSI): 1

J8:
  3V3  (1) (2)  5V
  GPIO2 (3) (4)  5V
  GPIO3 (5) (6)  GND
  GPIO4 (7) (8)  GPIO14
  GND   (9) (10) GPIO15
  GPIO17 (11) (12) GPIO18
  GPIO27 (13) (14) GND
  GPIO22 (15) (16) GPIO23
  3V3   (17) (18) GPIO24
  GPIO10 (19) (20) GND
  GPIO9  (21) (22) GPIO25
  GPIO11 (23) (24) GPIO8
  GND    (25) (26) GPIO7
  GPIO0  (27) (28) GPIO1
  GPIO5  (29) (30) GND
  GPIO6  (31) (32) GPIO12
  GPIO13 (33) (34) GND
  GPIO19 (35) (36) GPIO16
  GPIO26 (37) (38) GPIO20
  GND    (39) (40) GPIO21

For further information, please refer to https://pinout.xyz/
pi@raspberrypi:~ $

```

## Synopsis

```
pinout [-h] [-r REVISION] [-c] [-m]
```

## Description

A utility for querying Raspberry Pi GPIO pin-out information. Running **pinout** on its own will output a board diagram, and GPIO header diagram for the current Raspberry Pi. It is also possible to manually specify a revision of Pi, or (by [Configuring Remote GPIO](#) (page 35)) to output information about a remote Pi.

## Options

- h, --help**  
show this help message and exit
- r REVISION, --revision REVISION**  
RPi revision. Default is to autodetect revision of current device
- c, --color**  
Force colored output (by default, the output will include ANSI color codes if run in a color-capable terminal). See also [--monochrome](#) (page 56)
- m, --monochrome**  
Force monochrome output. See also [--color](#) (page 56)

## Examples

To output information about the current Raspberry Pi:

```
$ pinout
```

For a Raspberry Pi model 3B, this will output something like the following:

```

,-----,
| ooooooooooooooooooooo J8      +====
| 1oooooooooooooooooooo        | USB
|                               +====
|      Pi Model 3B V1.1        |
|      +----+                  +====
| |D| |SoC|                    | USB
| |S| |   |                    +====
| |I| +----+                  |
|                               |
|                               |C| +=====
|                               |S| | Net
| pwr          |HDMI| |I| |A| +=====
`-| |-----| |----|V|-----'

Revision          : a02082
SoC               : BCM2837
RAM               : 1024Mb
Storage           : MicroSD
USB ports         : 4 (excluding power)
Ethernet ports    : 1
Wi-fi             : True
Bluetooth         : True
Camera ports (CSI) : 1
Display ports (DSI) : 1

J8:
```

3V3	(1)	(2)	5V
GPIO2	(3)	(4)	5V
GPIO3	(5)	(6)	GND
GPIO4	(7)	(8)	GPIO14
GND	(9)	(10)	GPIO15
GPIO17	(11)	(12)	GPIO18
GPIO27	(13)	(14)	GND
GPIO22	(15)	(16)	GPIO23
3V3	(17)	(18)	GPIO24
GPIO10	(19)	(20)	GND
GPIO9	(21)	(22)	GPIO25
GPIO11	(23)	(24)	GPIO8
GND	(25)	(26)	GPIO7
GPIO0	(27)	(28)	GPIO1
GPIO5	(29)	(30)	GND
GPIO6	(31)	(32)	GPIO12
GPIO13	(33)	(34)	GND
GPIO19	(35)	(36)	GPIO16
GPIO26	(37)	(38)	GPIO20
GND	(39)	(40)	GPIO21

By default, if stdout is a console that supports color, ANSI codes will be used to produce color output. Output can be forced to be `--monochrome` (page 56):

```
$ pinout --monochrome
```

Or forced to be `--color` (page 56), in case you are redirecting to something capable of supporting ANSI codes:

```
$ pinout --color | less -SR
```

To manually specify the revision of Pi you want to query, use `--revision` (page 56). The tool understands both old-style [revision codes](http://elinux.org/RPi_HardwareHistory)<sup>33</sup> (such as for the model B):

```
$ pinout -r 000d
```

Or new-style [revision codes](http://elinux.org/RPi_HardwareHistory)<sup>34</sup> (such as for the Pi Zero W):

```
$ pinout -r 9000c1
```

<sup>33</sup> [http://elinux.org/RPi\\_HardwareHistory](http://elinux.org/RPi_HardwareHistory)

<sup>34</sup> [http://elinux.org/RPi\\_HardwareHistory](http://elinux.org/RPi_HardwareHistory)

```

pi@raspberrypi: ~
File Edit Tabs Help
pi@raspberrypi:~ $ pinout
-----
| 00000000000000000000 J8 |
| 10000000000000000000 | C
+---+ +---+ PiZero W | S
sd | | SoC | V1.1 | i
+---+ |hdmi| +---+ |usb| pwr |
+---+ +---+ +---+ +---+
Revision          : 9000c1
SoC               : BCM2835
RAM              : 512Mb
Storage          : MicroSD
USB ports        : 1 (excluding power)
Ethernet ports   : 0
Wi-fi            : True
Bluetooth        : True
Camera ports (CSI) : 1
Display ports (DSI): 0

J8:
  3V3  (1) (2)  5V
  GPIO2 (3) (4)  5V
  GPIO3 (5) (6)  GND
  GPIO4 (7) (8)  GPIO14
    GND (9) (10) GPIO15
  GPIO17 (11) (12) GPIO18
  GPIO27 (13) (14) GND
  GPIO22 (15) (16) GPIO23
    3V3 (17) (18) GPIO24
  GPIO10 (19) (20) GND
  GPIO9  (21) (22) GPIO25
  GPIO11 (23) (24) GPIO8
    GND (25) (26) GPIO7
  GPIO0  (27) (28) GPIO1
  GPIO5  (29) (30) GND
  GPIO6  (31) (32) GPIO12
  GPIO13 (33) (34) GND
  GPIO19 (35) (36) GPIO16
  GPIO26 (37) (38) GPIO20
    GND (39) (40) GPIO21

For further information, please refer to https://pinout.xyz/
pi@raspberrypi:~ $

```

You can also use the tool with *Configuring Remote GPIO* (page 35) to query remote Raspberry Pi's:

```
$ GPIOZERO_PIN_FACTORY=pigpio PIGPIO_ADDR=other_pi pinout
```

Or run the tool directly on a PC using the mock pin implementation (although in this case you'll almost certainly want to specify the Pi revision manually):

```
$ GPIOZERO_PIN_FACTORY=mock pinout -r a22042
```

## Environment Variables

**GPIOZERO\_PIN\_FACTORY** The library to use when communicating with the GPIO pins. Defaults to attempting to load RPi.GPIO, then RPIO, then pigpio, and finally uses a native Python implementation. Valid values include “rpigpio”, “rpio”, “pigpio”, “native”, and “mock”. The latter is most useful on non-Pi platforms as it emulates a Raspberry Pi model 3B (by default).

**PIGPIO\_ADDR** The hostname of the Raspberry Pi the pigpio library should attempt to connect to (if the pigpio pin factory is being used). Defaults to `localhost`.

**PIGPIO\_PORT** The port number the pigpio library should attempt to connect to (if the pigpio pin factory is being used). Defaults to `8888`.





---

## Frequently Asked Questions

---

### How do I keep my 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()`<sup>35</sup> 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()
```

---

<sup>35</sup> <https://docs.python.org/3.5/library/signal.html#signal.pause>

## My event handler isn't being called?

When assigning event handlers, don't call the function you're assigning. For example:

```
from gpiozero import Button

def pushed():
    print("Don't push the button!")

b = Button(17)
b.when_pressed = pushed()
```

In the case above, when assigning to `when_pressed`, the thing that is assigned is the *result of calling* the `pushed` function. Because `pushed` doesn't explicitly return anything, the result is `None`. Hence this is equivalent to doing:

```
b.when_pressed = None
```

This doesn't raise an error because it's perfectly valid: it's what you assign when you don't want the event handler to do anything. Instead, you want to do the following:

```
b.when_pressed = pushed
```

This will assign the function to the event handler *without calling it*. This is the crucial difference between `my_function` (a reference to a function) and `my_function()` (the result of calling a function).

## Why do I get PinFactoryFallback warnings when I import gpiozero?

You are most likely working in a virtual Python environment and have forgotten to install a pin driver library like `RPi.GPIO`. GPIO Zero relies upon lower level pin drivers to handle interfacing to the GPIO pins on the Raspberry Pi, so you can eliminate the warning simply by installing GPIO Zero's first preference:

```
$ pip install rpi.gpio
```

When GPIO Zero is imported it attempts to find a pin driver by importing them in a preferred order (detailed in [API - Pins](#) (page 163)). If it fails to load its first preference (`RPi.GPIO`) it notifies you with a warning, then falls back to trying its second preference and so on. Eventually it will fall back all the way to the `native` implementation. This is a pure Python implementation built into GPIO Zero itself. While this will work for most things it's almost certainly not what you want (it doesn't support PWM, and it's quite slow at certain things).

If you want to use a pin driver other than the default, and you want to suppress the warnings you've got a couple of options:

1. Explicitly specify what pin driver you want via an environment variable. For example:

```
$ GPIOZERO_PIN_FACTORY=pigpio python3
```

In this case no warning is issued because there's no fallback; either the specified factory loads or it fails in which case an `ImportError`<sup>36</sup> will be raised.

2. Suppress the warnings and let the fallback mechanism work:

```
>>> import warnings
>>> warnings.simplefilter('ignore')
>>> import gpiozero
```

Refer to the `warnings`<sup>37</sup> module documentation for more refined ways to filter out specific warning classes.

---

<sup>36</sup> <https://docs.python.org/3.5/library/exceptions.html#ImportError>

<sup>37</sup> <https://docs.python.org/3.5/library/warnings.html#module-warnings>

## 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 1.4.0 (/usr/lib/python3/dist-packages)]
>>> require('gpiozero')[0].version
'1.4.0'
```

If you have multiple versions installed (e.g. from `pip` and `apt`) 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 `pip`. This can be done with the following command in Raspbian:

```
$ sudo apt install python3-pip
```

Alternatively, install `pip` with `get-pip`<sup>38</sup>.

---

<sup>38</sup> <https://pip.pypa.io/en/stable/installing/>



Contributions to the library are welcome! Here are some guidelines to follow.

### Suggestions

Please make suggestions for additional components or enhancements to the codebase by opening an [issue](#)<sup>39</sup> explaining your reasoning clearly.

### Bugs

Please submit bug reports by opening an [issue](#)<sup>40</sup> explaining the problem clearly using code examples.

### Documentation

The documentation source lives in the [docs](#)<sup>41</sup> 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 which close (or intend to close) an issue should include the phrase “fix #123” or “close #123” where #123 is the issue number, as well as include a short description, for example: “Add LED example, close #123”, and pull requests should aim to match or closely match the corresponding issue title.

---

<sup>39</sup> <https://github.com/RPi-Distro/python-gpiozero/issues>

<sup>40</sup> <https://github.com/RPi-Distro/python-gpiozero/issues>

<sup>41</sup> <https://github.com/RPi-Distro/python-gpiozero/tree/master/docs>

## 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 2/3

The library is 100% compatible with both Python 2 and 3. We intend to drop Python 2 support in 2020 when Python 2 reaches [end-of-life](http://legacy.python.org/dev/peps/pep-0373/)<sup>42</sup>.

---

<sup>42</sup> <http://legacy.python.org/dev/peps/pep-0373/>

# CHAPTER 10

---

## Development

---

The main GitHub repository for the project can be found at:

<https://github.com/RPi-Distro/python-gpiozero>

For anybody wishing to hack on the project, we recommend starting off by getting to grips with some simple device classes. Pick something like *LED* (page 81) and follow its heritage backward to *DigitalOutputDevice* (page 92). Follow that back to *OutputDevice* (page 95) and you should have a good understanding of simple output devices along with a grasp of how GPIO Zero relies fairly heavily upon inheritance to refine the functionality of devices. The same can be done for input devices, and eventually more complex devices (composites and SPI based).

## Development installation

If you wish to develop GPIO Zero itself, we recommend obtaining the source by cloning the GitHub repository and then use the “develop” target of the Makefile which will install the package as a link to the cloned repository allowing in-place development (it also builds a tags file for use with vim/emacs with Exuberant’s ctags utility). The following example demonstrates this method within a virtual Python environment:

```
$ sudo apt install lsb-release build-essential git git-core \
> exuberant-ctags virtualenvwrapper python-virtualenv python3-virtualenv \
> python-dev python3-dev
$ cd
$ mkvirtualenv -p /usr/bin/python3 python-gpiozero
$ workon python-gpiozero
(python-gpiozero) $ git clone https://github.com/RPi-Distro/python-gpiozero.git
(python-gpiozero) $ cd python-gpiozero
(python-gpiozero) $ make develop
```

You will likely wish to install one or more pin implementations within the virtual environment (if you don’t, GPIO Zero will use the “native” pin implementation which is largely experimental at this stage and not very useful):

```
(python-gpiozero) $ pip install rpi.gpio pigpio
```

If you are working on SPI devices you may also wish to install the `spidev` package to provide hardware SPI capabilities (again, GPIO Zero will work without this, but a big-banging software SPI implementation will be used instead):

```
(python-gpiozero) $ pip install spidev
```

To pull the latest changes from git into your clone and update your installation:

```
$ workon python-gpiozero
(python-gpiozero) $ cd ~/python-gpiozero
(python-gpiozero) $ git pull
(python-gpiozero) $ make develop
```

To remove your installation, destroy the sandbox and the clone:

```
(python-gpiozero) $ deactivate
$ rmvirtualenv python-gpiozero
$ rm -fr ~/python-gpiozero
```

## Building the docs

If you wish to build the docs, you'll need a few more dependencies. Inkscape is used for conversion of SVGs to other formats, Graphviz is used for rendering certain charts, and TeX Live is required for building PDF output. The following command should install all required dependencies:

```
$ sudo apt install texlive-latex-recommended texlive-latex-extra \
    texlive-fonts-recommended graphviz inkscape
```

Once these are installed, you can use the “doc” target to build the documentation:

```
$ workon python-gpiozero
(python-gpiozero) $ cd ~/python-gpiozero
(python-gpiozero) $ make doc
```

The HTML output is written to docs/\_build/html while the PDF output goes to docs/\_build/latex.

## Test suite

If you wish to run the GPIO Zero test suite, follow the instructions in *Development installation* (page 67) above and then make the “test” target within the sandbox:

```
$ workon python-gpiozero
(python-gpiozero) $ cd ~/python-gpiozero
(python-gpiozero) $ make test
```

The test suite expects pins 22 and 27 (by default) to be wired together in order to run the “real” pin tests. The pins used by the test suite can be overridden with the environment variables GPIOZERO\_TEST\_PIN (defaults to 22) and GPIOZERO\_TEST\_INPUT\_PIN (defaults to 27).

**Warning:** When wiring GPIOs together, ensure a load (like a 330Ω resistor) is placed between them. Failure to do so may lead to blown GPIO pins (your humble author has a fried GPIO27 as a result of such laziness, although it did take *many* runs of the test suite before this occurred!).



# CHAPTER 11

---

## API - 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 *Basic Recipes* (page 3) page for more information.

---

### Button

**class** `gpiozero.Button` (*pin*, \*, *pull\_up=True*, *bounce\_time=None*, *hold\_time=1*, *hold\_repeat=False*,  
*pin\_factory=None*)

Extends *DigitalInputDevice* (page 77) 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* (page 69) 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*<sup>43</sup>) – The GPIO pin which the button is attached to. See *Pin Numbering* (page 3) for valid pin numbers.
- **pull\_up** (*bool*<sup>44</sup>) – 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.

---

<sup>43</sup> <https://docs.python.org/3.5/library/functions.html#int>

<sup>44</sup> <https://docs.python.org/3.5/library/functions.html#bool>

- **bounce\_time** (*float*<sup>45</sup>) – 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*<sup>46</sup>) – The length of time (in seconds) to wait after the button is pushed, until executing the *when\_held* (page 70) handler. Defaults to 1.
- **hold\_repeat** (*bool*<sup>47</sup>) – If `True`, the *when\_held* (page 70) handler will be repeatedly executed as long as the device remains active, every *hold\_time* seconds. If `False` (the default) the *when\_held* (page 70) handler will be only be executed once per hold.
- **pin\_factory** (*Factory* (page 166)) – See *API - Pins* (page 163) for more information (this is an advanced feature which most users can ignore).

**wait\_for\_press** (*timeout=None*)

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

**Parameters** **timeout** (*float*<sup>48</sup>) – 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*<sup>49</sup>) – 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* (page 70) event rather than when the device activated, in contrast to *active\_time* (page 149). If the device is not currently held, this is `None`.

**hold\_repeat**

If `True`, *when\_held* (page 70) will be executed repeatedly with *hold\_time* (page 70) seconds between each invocation.

**hold\_time**

The length of time (in seconds) to wait after the device is activated, until executing the *when\_held* (page 70) handler. If *hold\_repeat* (page 70) is `True`, this is also the length of time between invocations of *when\_held* (page 70).

**is\_held**

When `True`, the device has been active for at least *hold\_time* (page 70) 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* (page 167) 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* (page 70) seconds.

---

<sup>45</sup> <https://docs.python.org/3.5/library/functions.html#float>

<sup>46</sup> <https://docs.python.org/3.5/library/functions.html#float>

<sup>47</sup> <https://docs.python.org/3.5/library/functions.html#bool>

<sup>48</sup> <https://docs.python.org/3.5/library/functions.html#float>

<sup>49</sup> <https://docs.python.org/3.5/library/functions.html#float>

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*, \*, *queue\_len*=5, *sample\_rate*=100, *threshold*=0.5, *partial*=False, *pin\_factory*=None)

Extends `SmoothedInputDevice` (page 78) and represents a single pin line sensor like the TCRT5000 infra-red proximity sensor found in the [CamJam #3 EduKit](http://camjam.me/?page_id=1035)<sup>50</sup>.

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*<sup>51</sup>) – The GPIO pin which the sensor is attached to. See [Pin Numbering](#) (page 3) for valid pin numbers.
- **queue\_len** (*int*<sup>52</sup>) – The length of the queue used to store values read from the sensor. This defaults to 5.
- **sample\_rate** (*float*<sup>53</sup>) – The number of values to read from the device (and append to the internal queue) per second. Defaults to 100.

<sup>50</sup> [http://camjam.me/?page\\_id=1035](http://camjam.me/?page_id=1035)

<sup>51</sup> <https://docs.python.org/3.5/library/functions.html#int>

<sup>52</sup> <https://docs.python.org/3.5/library/functions.html#int>

<sup>53</sup> <https://docs.python.org/3.5/library/functions.html#float>

- **threshold** (*float*<sup>54</sup>) – 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* (page 79) property, and all appropriate events will be fired.
- **partial** (*bool*<sup>55</sup>) – When `False` (the default), the object will not return a value for *is\_active* (page 79) until the internal queue has filled with values. Only set this to `True` if you require values immediately after object construction.
- **pin\_factory** (*Factory* (page 166)) – See *API - Pins* (page 163) for more information (this is an advanced feature which most users can ignore).

**wait\_for\_line** (*timeout=None*)

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

**Parameters** **timeout** (*float*<sup>56</sup>) – 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*<sup>57</sup>) – Number of seconds to wait before proceeding. If this is `None` (the default), then wait indefinitely until the device is active.

**pin**

The *Pin* (page 167) 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*, *pin\_factory=None*)

Extends *SmoothedInputDevice* (page 78) and represents a passive infra-red (PIR) motion sensor like the sort found in the *CamJam #2 EduKit*<sup>58</sup>.

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:

---

<sup>54</sup> <https://docs.python.org/3.5/library/functions.html#float>

<sup>55</sup> <https://docs.python.org/3.5/library/functions.html#bool>

<sup>56</sup> <https://docs.python.org/3.5/library/functions.html#float>

<sup>57</sup> <https://docs.python.org/3.5/library/functions.html#float>

<sup>58</sup> [http://camjam.me/?page\\_id=623](http://camjam.me/?page_id=623)

```
from gpiozero import MotionSensor

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

### Parameters

- **pin** (*int*<sup>59</sup>) – The GPIO pin which the sensor is attached to. See *Pin Numbering* (page 3) for valid pin numbers.
- **queue\_len** (*int*<sup>60</sup>) – 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*<sup>61</sup>) – The number of values to read from the device (and append to the internal queue) per second. Defaults to 100.
- **threshold** (*float*<sup>62</sup>) – 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* (page 79) property, and all appropriate events will be fired.
- **partial** (*bool*<sup>63</sup>) – When `False` (the default), the object will not return a value for *is\_active* (page 79) until the internal queue has filled with values. Only set this to `True` if you require values immediately after object construction.
- **pin\_factory** (*Factory* (page 166)) – See *API - Pins* (page 163) for more information (this is an advanced feature which most users can ignore).

**wait\_for\_motion** (*timeout=None*)

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

**Parameters** **timeout** (*float*<sup>64</sup>) – 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*<sup>65</sup>) – 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* (page 167) 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.

<sup>59</sup> <https://docs.python.org/3.5/library/functions.html#int>

<sup>60</sup> <https://docs.python.org/3.5/library/functions.html#int>

<sup>61</sup> <https://docs.python.org/3.5/library/functions.html#float>

<sup>62</sup> <https://docs.python.org/3.5/library/functions.html#float>

<sup>63</sup> <https://docs.python.org/3.5/library/functions.html#bool>

<sup>64</sup> <https://docs.python.org/3.5/library/functions.html#float>

<sup>65</sup> <https://docs.python.org/3.5/library/functions.html#float>

**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*, *pin\_factory=None*)

Extends `SmoothedInputDevice` (page 78) and represents a light dependent resistor (LDR).

Connect one leg of the LDR to the 3V3 pin; connect one leg of a 1 $\mu$ F 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*<sup>66</sup>) – The GPIO pin which the sensor is attached to. See *Pin Numbering* (page 3) for valid pin numbers.
- **queue\_len** (*int*<sup>67</sup>) – The length of the queue used to store values read from the circuit. This defaults to 5.
- **charge\_time\_limit** (*float*<sup>68</sup>) – 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 1 $\mu$ F capacitor coupled with the LDR from the *CamJam #2 EduKit*<sup>69</sup>. You may need to adjust this value for different valued capacitors or LDRs.
- **threshold** (*float*<sup>70</sup>) – 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*<sup>71</sup>) – When `False` (the default), the object will not return a value for *is\_active* (page 79) until the internal queue has filled with values. Only set this to `True` if you require values immediately after object construction.
- **pin\_factory** (*Factory* (page 166)) – See *API - Pins* (page 163) for more information (this is an advanced feature which most users can ignore).

**wait\_for\_dark** (*timeout=None*)

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

---

<sup>66</sup> <https://docs.python.org/3.5/library/functions.html#int>

<sup>67</sup> <https://docs.python.org/3.5/library/functions.html#int>

<sup>68</sup> <https://docs.python.org/3.5/library/functions.html#float>

<sup>69</sup> [http://camjam.me/?page\\_id=623](http://camjam.me/?page_id=623)

<sup>70</sup> <https://docs.python.org/3.5/library/functions.html#float>

<sup>71</sup> <https://docs.python.org/3.5/library/functions.html#bool>

**Parameters** `timeout` (*float*<sup>72</sup>) – 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*<sup>73</sup>) – 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* (page 167) 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*, *pin\_factory=None*)

Extends *SmoothedInputDevice* (page 78) and represents an HC-SR04 ultrasonic distance sensor, as found in the *CamJam #3 EduKit*<sup>74</sup>.

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
```

<sup>72</sup> <https://docs.python.org/3.5/library/functions.html#float>

<sup>73</sup> <https://docs.python.org/3.5/library/functions.html#float>

<sup>74</sup> [http://camjam.me/?page\\_id=1035](http://camjam.me/?page_id=1035)



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

### Parameters

- **echo** (*int*<sup>75</sup>) – The GPIO pin which the ECHO pin is attached to. See *Pin Numbering* (page 3) for valid pin numbers.
- **trigger** (*int*<sup>76</sup>) – The GPIO pin which the TRIG pin is attached to. See *Pin Numbering* (page 3) for valid pin numbers.
- **queue\_len** (*int*<sup>77</sup>) – The length of the queue used to store values read from the sensor. This defaults to 30.
- **max\_distance** (*float*<sup>78</sup>) – 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*<sup>79</sup>) – 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*<sup>80</sup>) – When `False` (the default), the object will not return a value for `is_active` (page 79) until the internal queue has filled with values. Only set this to `True` if you require values immediately after object construction.
- **pin\_factory** (*Factory* (page 166)) – See *API - Pins* (page 163) for more information (this is an advanced feature which most users can ignore).

**wait\_for\_in\_range** (*timeout=None*)

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

**Parameters** **timeout** (*float*<sup>81</sup>) – 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*<sup>82</sup>) – 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` (page 76).

### echo

Returns the *Pin* (page 167) 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` (page 76) is equal to `max_distance` (page 76), `value` will be 1.

### threshold\_distance

The distance, measured in meters, that will trigger the `when_in_range` (page 77) and

---

<sup>75</sup> <https://docs.python.org/3.5/library/functions.html#int>

<sup>76</sup> <https://docs.python.org/3.5/library/functions.html#int>

<sup>77</sup> <https://docs.python.org/3.5/library/functions.html#int>

<sup>78</sup> <https://docs.python.org/3.5/library/functions.html#float>

<sup>79</sup> <https://docs.python.org/3.5/library/functions.html#float>

<sup>80</sup> <https://docs.python.org/3.5/library/functions.html#bool>

<sup>81</sup> <https://docs.python.org/3.5/library/functions.html#float>

<sup>82</sup> <https://docs.python.org/3.5/library/functions.html#float>



`when_out_of_range` (page 77) events when crossed. This is simply a meter-scaled variant of the usual `threshold` attribute.

#### **trigger**

Returns the `Pin` (page 167) 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**

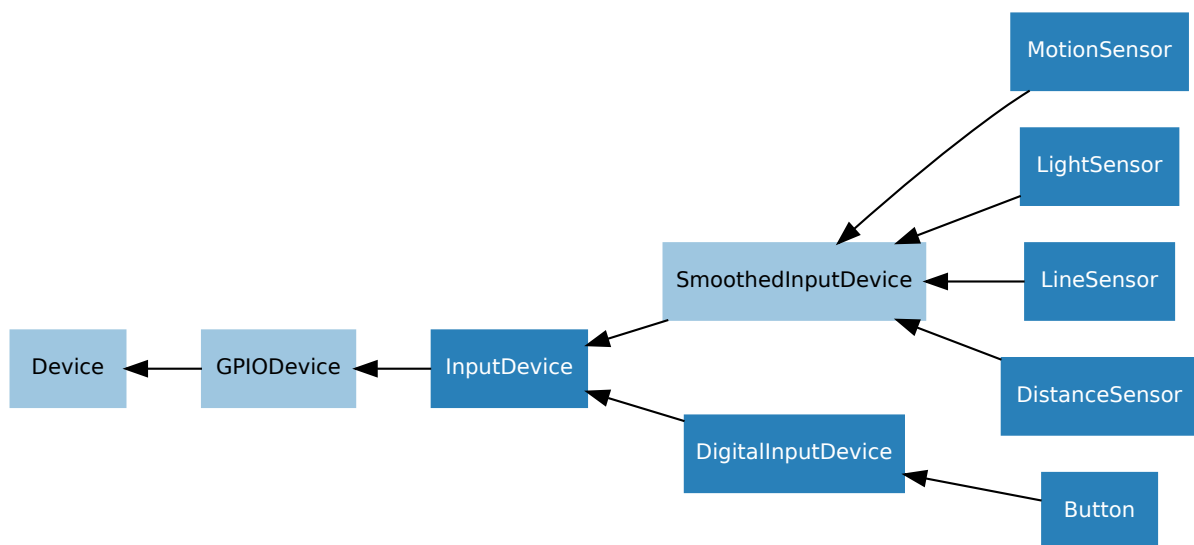
The function to run when the device changes state from inactive to active.

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 (abstract classes are shaded lighter than concrete classes):



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,
                                   pin_factory=None)
```

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

This class extends *InputDevice* (page 79) 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*<sup>83</sup>) – 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.
- **pin\_factory** (*Factory* (page 166)) – See *API - Pins* (page 163) for more information (this is an advanced feature which most users can ignore).

## SmoothedInputDevice

**class** `gpiozero.SmoothedInputDevice` (*pin*, \*, *pull\_up=False*, *threshold=0.5*, *queue\_len=5*, *sample\_wait=0.0*, *partial=False*, *pin\_factory=None*)

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

This class extends *InputDevice* (page 79) 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* (page 79) 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*<sup>84</sup>) – The value above which the device will be considered “on”.
- **queue\_len** (*int*<sup>85</sup>) – The length of the internal queue which is filled by the background thread.
- **sample\_wait** (*float*<sup>86</sup>) – 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*<sup>87</sup>) – If `False` (the default), attempts to read the state of the device (from the *is\_active* (page 79) 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.
- **pin\_factory** (*Factory* (page 166)) – See *API - Pins* (page 163) for more information (this is an advanced feature which most users can ignore).

**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.

---

<sup>83</sup> <https://docs.python.org/3.5/library/functions.html#float>

<sup>84</sup> <https://docs.python.org/3.5/library/functions.html#float>

<sup>85</sup> <https://docs.python.org/3.5/library/functions.html#int>

<sup>86</sup> <https://docs.python.org/3.5/library/functions.html#float>

<sup>87</sup> <https://docs.python.org/3.5/library/functions.html#bool>

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* (page 147) descendants can also be used as context managers using the `with`<sup>88</sup> 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* (page 79) or *is\_active* (page 79) 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* (page 79) exceeds this amount, then *is\_active* (page 79) will return `True`.

#### **value**

Returns the mean of the values in the internal queue. This is compared to *threshold* (page 79) to determine whether *is\_active* (page 79) is `True`.

## InputDevice

**class** `gpiozero.InputDevice` (*pin*, \*, *pull\_up=False*, *pin\_factory=None*)

Represents a generic GPIO input device.

This class extends *GPIODevice* (page 80) 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* (page 80) setting.

#### **Parameters**

- **pin** (*int*<sup>89</sup>) – The GPIO pin (in Broadcom numbering) that the device is connected to. If this is `None` a *GPIODeviceError* (page 178) will be raised.

<sup>88</sup> [https://docs.python.org/3.5/reference/compound\\_stmts.html#with](https://docs.python.org/3.5/reference/compound_stmts.html#with)

<sup>89</sup> <https://docs.python.org/3.5/library/functions.html#int>

- **pull\_up** (*bool*<sup>90</sup>) – If `True`, the pin will be pulled high with an internal resistor. If `False` (the default), the pin will be pulled low.
- **pin\_factory** (*Factory* (page 166)) – See *API - Pins* (page 163) for more information (this is an advanced feature which most users can ignore).

**pull\_up**

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

## GPIODevice

**class** `gpiozero.GPIODevice` (*pin*, *pin\_factory=None*)

Extends *Device* (page 147). Represents a generic GPIO device and provides the services common to all single-pin GPIO devices (like ensuring two GPIO devices do not share a *pin* (page 80)).

**Parameters** *pin* (*int*<sup>91</sup>) – The GPIO pin (in BCM numbering) that the device is connected to. If this is `None`, *GPIOPinMissing* (page 178) will be raised. If the pin is already in use by another device, *GPIOPinInUse* (page 178) 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* (page 147) descendants can also be used as context managers using the `with`<sup>92</sup> statement. For example:

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

**pin**

The *Pin* (page 167) that the device is connected to. This will be `None` if the device has been closed (see the `close()` (page 80) method). When dealing with GPIO pins, query `pin.number` to discover the GPIO pin (in BCM numbering) that the device is connected to.

---

<sup>90</sup> <https://docs.python.org/3.5/library/functions.html#bool>

<sup>91</sup> <https://docs.python.org/3.5/library/functions.html#int>

<sup>92</sup> [https://docs.python.org/3.5/reference/compound\\_stmts.html#with](https://docs.python.org/3.5/reference/compound_stmts.html#with)

# CHAPTER 12

---

## API - 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 *Basic Recipes* (page 3) page for more information.

---

### LED

**class** `gpiozero.LED` (*pin*, \*, *active\_high=True*, *initial\_value=False*, *pin\_factory=None*)

Extends *DigitalOutputDevice* (page 92) 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*<sup>93</sup>) – The GPIO pin which the LED is attached to. See *Pin Numbering* (page 3) for valid pin numbers.
- **active\_high** (*bool*<sup>94</sup>) – 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).

---

<sup>93</sup> <https://docs.python.org/3.5/library/functions.html#int>

<sup>94</sup> <https://docs.python.org/3.5/library/functions.html#bool>

- **initial\_value** (*bool*<sup>95</sup>) – 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.
- **pin\_factory** (*Factory* (page 166)) – See *API - Pins* (page 163) for more information (this is an advanced feature which most users can ignore).

**blink** (*on\_time=1, off\_time=1, n=None, background=True*)

Make the device turn on and off repeatedly.

#### Parameters

- **on\_time** (*float*<sup>96</sup>) – Number of seconds on. Defaults to 1 second.
- **off\_time** (*float*<sup>97</sup>) – Number of seconds off. Defaults to 1 second.
- **n** (*int*<sup>98</sup>) – Number of times to blink; `None` (the default) means forever.
- **background** (*bool*<sup>99</sup>) – 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* (page 167) 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*,  
*pin\_factory=None*)

Extends *PWMOutputDevice* (page 93) 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*<sup>100</sup>) – The GPIO pin which the LED is attached to. See *Pin Numbering* (page 3) for valid pin numbers.
- **active\_high** (*bool*<sup>101</sup>) – If `True` (the default), the `on()` (page 83) method will set the GPIO to HIGH. If `False`, the `on()` (page 83) method will set the GPIO to LOW (the `off()` (page 83) method always does the opposite).

---

<sup>95</sup> <https://docs.python.org/3.5/library/functions.html#bool>

<sup>96</sup> <https://docs.python.org/3.5/library/functions.html#float>

<sup>97</sup> <https://docs.python.org/3.5/library/functions.html#float>

<sup>98</sup> <https://docs.python.org/3.5/library/functions.html#int>

<sup>99</sup> <https://docs.python.org/3.5/library/functions.html#bool>

<sup>100</sup> <https://docs.python.org/3.5/library/functions.html#int>

<sup>101</sup> <https://docs.python.org/3.5/library/functions.html#bool>

- **initial\_value** (*float*<sup>102</sup>) – 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*<sup>103</sup>) – The frequency (in Hz) of pulses emitted to drive the LED. Defaults to 100Hz.
- **pin\_factory** (`Factory` (page 166)) – See *API - Pins* (page 163) for more information (this is an advanced feature which most users can ignore).

**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*<sup>104</sup>) – Number of seconds on. Defaults to 1 second.
- **off\_time** (*float*<sup>105</sup>) – Number of seconds off. Defaults to 1 second.
- **fade\_in\_time** (*float*<sup>106</sup>) – Number of seconds to spend fading in. Defaults to 0.
- **fade\_out\_time** (*float*<sup>107</sup>) – Number of seconds to spend fading out. Defaults to 0.
- **n** (*int*<sup>108</sup>) – Number of times to blink; `None` (the default) means forever.
- **background** (*bool*<sup>109</sup>) – 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*<sup>110</sup>) – Number of seconds to spend fading in. Defaults to 1.
- **fade\_out\_time** (*float*<sup>111</sup>) – Number of seconds to spend fading out. Defaults to 1.
- **n** (*int*<sup>112</sup>) – Number of times to pulse; `None` (the default) means forever.
- **background** (*bool*<sup>113</sup>) – 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* (page 84) is 0.0), this changes it to “fully” on (*value* (page 84) is 1.0). If the device has a duty cycle (*value* (page 84)) of 0.1, this will toggle it to 0.9, and so on.

<sup>102</sup> <https://docs.python.org/3.5/library/functions.html#float>

<sup>103</sup> <https://docs.python.org/3.5/library/functions.html#int>

<sup>104</sup> <https://docs.python.org/3.5/library/functions.html#float>

<sup>105</sup> <https://docs.python.org/3.5/library/functions.html#float>

<sup>106</sup> <https://docs.python.org/3.5/library/functions.html#float>

<sup>107</sup> <https://docs.python.org/3.5/library/functions.html#float>

<sup>108</sup> <https://docs.python.org/3.5/library/functions.html#int>

<sup>109</sup> <https://docs.python.org/3.5/library/functions.html#bool>

<sup>110</sup> <https://docs.python.org/3.5/library/functions.html#float>

<sup>111</sup> <https://docs.python.org/3.5/library/functions.html#float>

<sup>112</sup> <https://docs.python.org/3.5/library/functions.html#int>

<sup>113</sup> <https://docs.python.org/3.5/library/functions.html#bool>

**is\_lit**

Returns `True` if the device is currently active (*value* (page 84) is non-zero) and `False` otherwise.

**pin**

The *Pin* (page 167) 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, pin\_factory=None*)

Extends *Device* (page 147) 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*<sup>114</sup>) – The GPIO pin that controls the red component of the RGB LED.
- **green** (*int*<sup>115</sup>) – The GPIO pin that controls the green component of the RGB LED.
- **blue** (*int*<sup>116</sup>) – The GPIO pin that controls the blue component of the RGB LED.
- **active\_high** (*bool*<sup>117</sup>) – 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*<sup>118</sup>) – The initial color for the RGB LED. Defaults to black `(0, 0, 0)`.
- **pwm** (*bool*<sup>119</sup>) – If `True` (the default), construct *PWMLED* (page 82) instances for each component of the RGBLED. If `False`, construct regular *LED* (page 81) instances, which prevents smooth color graduations.
- **pin\_factory** (*Factory* (page 166)) – See *API - Pins* (page 163) for more information (this is an advanced feature which most users can ignore).

**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

---

<sup>114</sup> <https://docs.python.org/3.5/library/functions.html#int>

<sup>115</sup> <https://docs.python.org/3.5/library/functions.html#int>

<sup>116</sup> <https://docs.python.org/3.5/library/functions.html#int>

<sup>117</sup> <https://docs.python.org/3.5/library/functions.html#bool>

<sup>118</sup> <https://docs.python.org/3.5/library/stdtypes.html#tuple>

<sup>119</sup> <https://docs.python.org/3.5/library/functions.html#bool>



- **on\_time** (*float*<sup>120</sup>) – Number of seconds on. Defaults to 1 second.
- **off\_time** (*float*<sup>121</sup>) – Number of seconds off. Defaults to 1 second.
- **fade\_in\_time** (*float*<sup>122</sup>) – Number of seconds to spend fading in. Defaults to 0. Must be 0 if `pwm` was `False` when the class was constructed (`ValueError`<sup>123</sup> will be raised if not).
- **fade\_out\_time** (*float*<sup>124</sup>) – Number of seconds to spend fading out. Defaults to 0. Must be 0 if `pwm` was `False` when the class was constructed (`ValueError`<sup>125</sup> will be raised if not).
- **on\_color** (*tuple*<sup>126</sup>) – The color to use when the LED is “on”. Defaults to white.
- **off\_color** (*tuple*<sup>127</sup>) – The color to use when the LED is “off”. Defaults to black.
- **n** (*int*<sup>128</sup>) – Number of times to blink; `None` (the default) means forever.
- **background** (*bool*<sup>129</sup>) – 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*<sup>130</sup>) – Number of seconds to spend fading in. Defaults to 1.
- **fade\_out\_time** (*float*<sup>131</sup>) – Number of seconds to spend fading out. Defaults to 1.
- **on\_color** (*tuple*<sup>132</sup>) – The color to use when the LED is “on”. Defaults to white.
- **off\_color** (*tuple*<sup>133</sup>) – The color to use when the LED is “off”. Defaults to black.
- **n** (*int*<sup>134</sup>) – Number of times to pulse; `None` (the default) means forever.
- **background** (*bool*<sup>135</sup>) – 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.

<sup>120</sup> <https://docs.python.org/3.5/library/functions.html#float>

<sup>121</sup> <https://docs.python.org/3.5/library/functions.html#float>

<sup>122</sup> <https://docs.python.org/3.5/library/functions.html#float>

<sup>123</sup> <https://docs.python.org/3.5/library/exceptions.html#ValueError>

<sup>124</sup> <https://docs.python.org/3.5/library/functions.html#float>

<sup>125</sup> <https://docs.python.org/3.5/library/exceptions.html#ValueError>

<sup>126</sup> <https://docs.python.org/3.5/library/stdtypes.html#tuple>

<sup>127</sup> <https://docs.python.org/3.5/library/stdtypes.html#tuple>

<sup>128</sup> <https://docs.python.org/3.5/library/functions.html#int>

<sup>129</sup> <https://docs.python.org/3.5/library/functions.html#bool>

<sup>130</sup> <https://docs.python.org/3.5/library/functions.html#float>

<sup>131</sup> <https://docs.python.org/3.5/library/functions.html#float>

<sup>132</sup> <https://docs.python.org/3.5/library/stdtypes.html#tuple>

<sup>133</sup> <https://docs.python.org/3.5/library/stdtypes.html#tuple>

<sup>134</sup> <https://docs.python.org/3.5/library/functions.html#int>

<sup>135</sup> <https://docs.python.org/3.5/library/functions.html#bool>

**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`, *pin\_factory*=`None`)

Extends *DigitalOutputDevice* (page 92) 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*<sup>136</sup>) – The GPIO pin which the buzzer is attached to. See *Pin Numbering* (page 3) for valid pin numbers.
- **active\_high** (*bool*<sup>137</sup>) – 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*<sup>138</sup>) – 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.
- **pin\_factory** (*Factory* (page 166)) – See *API - Pins* (page 163) for more information (this is an advanced feature which most users can ignore).

**beep** (*on\_time*=1, *off\_time*=1, *n*=`None`, *background*=`True`)

Make the device turn on and off repeatedly.

**Parameters**

- **on\_time** (*float*<sup>139</sup>) – Number of seconds on. Defaults to 1 second.
- **off\_time** (*float*<sup>140</sup>) – Number of seconds off. Defaults to 1 second.
- **n** (*int*<sup>141</sup>) – Number of times to blink; `None` (the default) means forever.
- **background** (*bool*<sup>142</sup>) – 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).

---

<sup>136</sup> <https://docs.python.org/3.5/library/functions.html#int>

<sup>137</sup> <https://docs.python.org/3.5/library/functions.html#bool>

<sup>138</sup> <https://docs.python.org/3.5/library/functions.html#bool>

<sup>139</sup> <https://docs.python.org/3.5/library/functions.html#float>

<sup>140</sup> <https://docs.python.org/3.5/library/functions.html#float>

<sup>141</sup> <https://docs.python.org/3.5/library/functions.html#int>

<sup>142</sup> <https://docs.python.org/3.5/library/functions.html#bool>

**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` (page 167) 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, pin\_factory=None*)

Extends `CompositeDevice` (page 138) and represents a generic motor connected to a bi-directional motor driver circuit (i.e. an `H-bridge`<sup>143</sup>).

Attach an `H-bridge`<sup>144</sup> 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`<sup>145</sup>) – The GPIO pin that the forward input of the motor driver chip is connected to.
- **backward** (`int`<sup>146</sup>) – The GPIO pin that the backward input of the motor driver chip is connected to.
- **pwm** (`bool`<sup>147</sup>) – If `True` (the default), construct `PWMOutputDevice` (page 93) instances for the motor controller pins, allowing both direction and variable speed control. If `False`, construct `DigitalOutputDevice` (page 92) instances, allowing only direction control.
- **pin\_factory** (`Factory` (page 166)) – See *API - Pins* (page 163) for more information (this is an advanced feature which most users can ignore).

**backward** (*speed=1*)

Drive the motor backwards.

**Parameters** **speed** (`float`<sup>148</sup>) – 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).

<sup>143</sup> [https://en.wikipedia.org/wiki/H\\_bridge](https://en.wikipedia.org/wiki/H_bridge)

<sup>144</sup> [https://en.wikipedia.org/wiki/H\\_bridge](https://en.wikipedia.org/wiki/H_bridge)

<sup>145</sup> <https://docs.python.org/3.5/library/functions.html#int>

<sup>146</sup> <https://docs.python.org/3.5/library/functions.html#int>

<sup>147</sup> <https://docs.python.org/3.5/library/functions.html#bool>

<sup>148</sup> <https://docs.python.org/3.5/library/functions.html#float>

**forward** (*speed=1*)

Drive the motor forwards.

**Parameters** **speed** (*float*<sup>149</sup>) – 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, pin\_factory=None*)

Extends [CompositeDevice](#) (page 138) 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*<sup>150</sup>) – The GPIO pin which the device is attached to. See [Pin Numbering](#) (page 3) for valid pin numbers.
- **initial\_value** (*float*<sup>151</sup>) – 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](#) (page 89)).
- **min\_pulse\_width** (*float*<sup>152</sup>) – The pulse width corresponding to the servo’s minimum position. This defaults to 1ms.
- **max\_pulse\_width** (*float*<sup>153</sup>) – The pulse width corresponding to the servo’s maximum position. This defaults to 2ms.
- **frame\_width** (*float*<sup>154</sup>) – The length of time between servo control pulses measured in seconds. This defaults to 20ms which is a common value for servos.
- **pin\_factory** ([Factory](#) (page 166)) – See [API - Pins](#) (page 163) for more information (this is an advanced feature which most users can ignore).

---

<sup>149</sup> <https://docs.python.org/3.5/library/functions.html#float>

<sup>150</sup> <https://docs.python.org/3.5/library/functions.html#int>

<sup>151</sup> <https://docs.python.org/3.5/library/functions.html#float>

<sup>152</sup> <https://docs.python.org/3.5/library/functions.html#float>

<sup>153</sup> <https://docs.python.org/3.5/library/functions.html#float>

<sup>154</sup> <https://docs.python.org/3.5/library/functions.html#float>

**detach()**

Temporarily disable control of the servo. This is equivalent to setting *value* (page 89) 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* (page 89).

**source\_delay**

The delay (measured in seconds) in the loop used to read values from *source* (page 89). 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, pin_factory=None)
```

Extends *Servo* (page 88) 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* (page 88) 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* (page 89) 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*<sup>155</sup>) – The GPIO pin which the device is attached to. See *Pin Numbering* (page 3) for valid pin numbers.
- **initial\_angle** (*float*<sup>156</sup>) – 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* (page 91)).
- **min\_angle** (*float*<sup>157</sup>) – 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*<sup>158</sup>) – 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*<sup>159</sup>) – The pulse width corresponding to the servo’s minimum position. This defaults to 1ms.
- **max\_pulse\_width** (*float*<sup>160</sup>) – The pulse width corresponding to the servo’s maximum position. This defaults to 2ms.
- **frame\_width** (*float*<sup>161</sup>) – The length of time between servo control pulses measured in seconds. This defaults to 20ms which is a common value for servos.
- **pin\_factory** (*Factory* (page 166)) – See *API - Pins* (page 163) for more information (this is an advanced feature which most users can ignore).

### **detach()**

Temporarily disable control of the servo. This is equivalent to setting *value* (page 91) 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.

---

<sup>155</sup> <https://docs.python.org/3.5/library/functions.html#int>

<sup>156</sup> <https://docs.python.org/3.5/library/functions.html#float>

<sup>157</sup> <https://docs.python.org/3.5/library/functions.html#float>

<sup>158</sup> <https://docs.python.org/3.5/library/functions.html#float>

<sup>159</sup> <https://docs.python.org/3.5/library/functions.html#float>

<sup>160</sup> <https://docs.python.org/3.5/library/functions.html#float>

<sup>161</sup> <https://docs.python.org/3.5/library/functions.html#float>

**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()* (page 90) 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()* (page 90) 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* (page 91).

**source\_delay**

The delay (measured in seconds) in the loop used to read values from *source* (page 91). Defaults to 0.01 seconds which is generally sufficient to keep CPU usage to a minimum while providing adequate responsiveness.

**value**

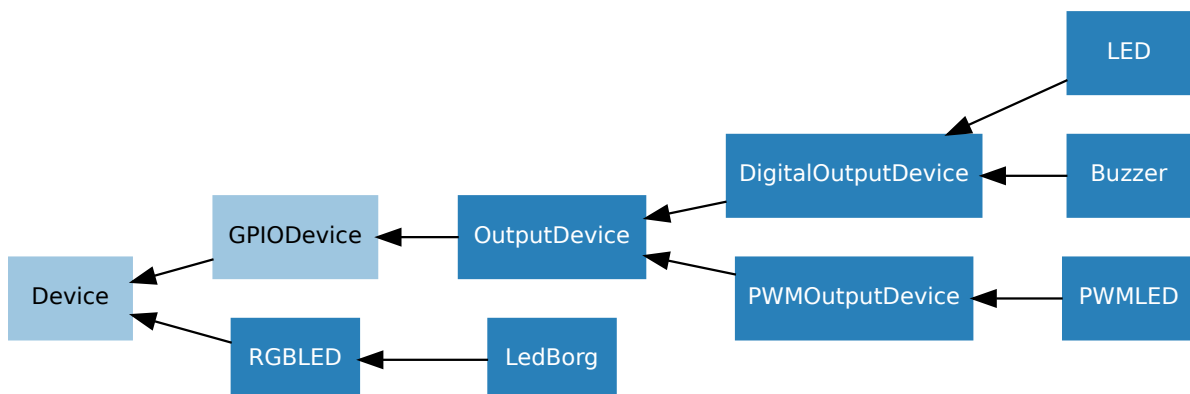
Represents the position of the servo as a value between -1 (the minimum position) and +1 (the maximum position). This can also be the special value `None` indicating that the servo is currently “uncontrolled”, i.e. that no control signal is being sent. Typically this means the servo’s position remains 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 (abstract classes are shaded lighter than concrete classes):



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, pin\_factory=None)

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

This class extends *OutputDevice* (page 95) with a *blink()* (page 92) 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*<sup>162</sup>) – Number of seconds on. Defaults to 1 second.
- **off\_time** (*float*<sup>163</sup>) – Number of seconds off. Defaults to 1 second.
- **n** (*int*<sup>164</sup>) – Number of times to blink; None (the default) means forever.
- **background** (*bool*<sup>165</sup>) – 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* (page 147) descendants can also be used as context managers using the *with*<sup>166</sup> statement. For example:

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

---

<sup>162</sup> <https://docs.python.org/3.5/library/functions.html#float>

<sup>163</sup> <https://docs.python.org/3.5/library/functions.html#float>

<sup>164</sup> <https://docs.python.org/3.5/library/functions.html#int>

<sup>165</sup> <https://docs.python.org/3.5/library/functions.html#bool>

<sup>166</sup> [https://docs.python.org/3.5/reference/compound\\_stmts.html#with](https://docs.python.org/3.5/reference/compound_stmts.html#with)



```
... 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*,  
*pin\_factory=None*)

Generic output device configured for pulse-width modulation (PWM).

### Parameters

- **pin** (*int*<sup>167</sup>) – The GPIO pin which the device is attached to. See *Pin Numbering* (page 3) for valid pin numbers.
- **active\_high** (*bool*<sup>168</sup>) – If `True` (the default), the `on()` (page 94) method will set the GPIO to HIGH. If `False`, the `on()` (page 94) method will set the GPIO to LOW (the `off()` (page 94) method always does the opposite).
- **initial\_value** (*float*<sup>169</sup>) – 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*<sup>170</sup>) – The frequency (in Hz) of pulses emitted to drive the device. Defaults to 100Hz.
- **pin\_factory** (*Factory* (page 166)) – See *API - Pins* (page 163) for more information (this is an advanced feature which most users can ignore).

**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*<sup>171</sup>) – Number of seconds on. Defaults to 1 second.
- **off\_time** (*float*<sup>172</sup>) – Number of seconds off. Defaults to 1 second.
- **fade\_in\_time** (*float*<sup>173</sup>) – Number of seconds to spend fading in. Defaults to 0.
- **fade\_out\_time** (*float*<sup>174</sup>) – Number of seconds to spend fading out. Defaults to 0.
- **n** (*int*<sup>175</sup>) – Number of times to blink; `None` (the default) means forever.
- **background** (*bool*<sup>176</sup>) – 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).

<sup>167</sup> <https://docs.python.org/3.5/library/functions.html#int>

<sup>168</sup> <https://docs.python.org/3.5/library/functions.html#bool>

<sup>169</sup> <https://docs.python.org/3.5/library/functions.html#float>

<sup>170</sup> <https://docs.python.org/3.5/library/functions.html#int>

<sup>171</sup> <https://docs.python.org/3.5/library/functions.html#float>

<sup>172</sup> <https://docs.python.org/3.5/library/functions.html#float>

<sup>173</sup> <https://docs.python.org/3.5/library/functions.html#float>

<sup>174</sup> <https://docs.python.org/3.5/library/functions.html#float>

<sup>175</sup> <https://docs.python.org/3.5/library/functions.html#int>

<sup>176</sup> <https://docs.python.org/3.5/library/functions.html#bool>

### `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* (page 147) descendants can also be used as context managers using the `with`<sup>177</sup> 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*<sup>178</sup>) – Number of seconds to spend fading in. Defaults to 1.
- **`fade_out_time`** (*float*<sup>179</sup>) – Number of seconds to spend fading out. Defaults to 1.
- **`n`** (*int*<sup>180</sup>) – Number of times to pulse; `None` (the default) means forever.
- **`background`** (*bool*<sup>181</sup>) – 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* (page 95) is 0.0), this changes it to “fully” on (*value* (page 95) is 1.0). If the device has a duty cycle (*value* (page 95)) of 0.1, this will toggle it to 0.9, and so on.

<sup>177</sup> [https://docs.python.org/3.5/reference/compound\\_stmts.html#with](https://docs.python.org/3.5/reference/compound_stmts.html#with)

<sup>178</sup> <https://docs.python.org/3.5/library/functions.html#float>

<sup>179</sup> <https://docs.python.org/3.5/library/functions.html#float>

<sup>180</sup> <https://docs.python.org/3.5/library/functions.html#int>

<sup>181</sup> <https://docs.python.org/3.5/library/functions.html#bool>

**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` (page 95) 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*, *pin\_factory=None*)

Represents a generic GPIO output device.

This class extends `GPIODevice` (page 80) to add facilities common to GPIO output devices: an `on()` (page 95) method to switch the device on, a corresponding `off()` (page 95) method, and a `toggle()` (page 95) method.

**Parameters**

- **pin** (*int*<sup>182</sup>) – The GPIO pin (in BCM numbering) that the device is connected to. If this is `None` a `GPIOPinMissing` (page 178) will be raised.
- **active\_high** (*bool*<sup>183</sup>) – If `True` (the default), the `on()` (page 95) method will set the GPIO to HIGH. If `False`, the `on()` (page 95) method will set the GPIO to LOW (the `off()` (page 95) method always does the opposite).
- **initial\_value** (*bool*<sup>184</sup>) – 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.
- **pin\_factory** (`Factory` (page 166)) – See *API - Pins* (page 163) for more information (this is an advanced feature which most users can ignore).

**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` (page 95) property is `True` when the device's pin is high. When `False` the `value` (page 95) 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` (page 95) (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.

<sup>182</sup> <https://docs.python.org/3.5/library/functions.html#int>

<sup>183</sup> <https://docs.python.org/3.5/library/functions.html#bool>

<sup>184</sup> <https://docs.python.org/3.5/library/functions.html#bool>

## GPIODevice

**class** gpiozero.GPIODevice (*pin*, \*, *pin\_factory*=None)

Extends *Device* (page 147). Represents a generic GPIO device and provides the services common to all single-pin GPIO devices (like ensuring two GPIO devices do not share a *pin* (page 80)).

**Parameters** *pin* (*int*<sup>185</sup>) – The GPIO pin (in BCM numbering) that the device is connected to. If this is None, *GPIOPinMissing* (page 178) will be raised. If the pin is already in use by another device, *GPIOPinInUse* (page 178) 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* (page 147) descendants can also be used as context managers using the *with*<sup>186</sup> statement. For example:

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

**pin**

The *Pin* (page 167) that the device is connected to. This will be None if the device has been closed (see the *close()* (page 80) method). When dealing with GPIO pins, query *pin.number* to discover the GPIO pin (in BCM numbering) that the device is connected to.

---

<sup>185</sup> <https://docs.python.org/3.5/library/functions.html#int>

<sup>186</sup> [https://docs.python.org/3.5/reference/compound\\_stmts.html#with](https://docs.python.org/3.5/reference/compound_stmts.html#with)

# CHAPTER 13

---

## API - SPI Devices

---

SPI stands for [Serial Peripheral Interface](https://en.wikipedia.org/wiki/Serial_Peripheral_Interface)<sup>187</sup> 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` (page 178) being raised. However, you can omit any arguments from either scheme. The defaults are:

---

<sup>187</sup> [https://en.wikipedia.org/wiki/Serial\\_Peripheral\\_Interface\\_Bus](https://en.wikipedia.org/wiki/Serial_Peripheral_Interface_Bus)

- *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.
- As with other GPIO based devices you can optionally specify a *pin\_factory* argument overriding the default pin factory (see [API - Pins](#) (page 163) for more information).

Hence the following constructors are all equivalent:

```
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**<sup>188</sup> is a 10-bit analog to digital converter with 1 channel. Please note that the MCP3001 always operates in differential mode, measuring the value of IN+ relative to IN-.

**value**

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

**class** gpiozero.**MCP3002** (*channel=0*, *differential=False*, \*\**spi\_args*)

The **MCP3002**<sup>189</sup> 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), the MCP3002/3202 have 2 channels (0-1), and the MCP3001/3201/3301 only have 1 channel.

**differential**

If True, the device is operated in 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** (page 99) 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 certain devices operating in differential mode).

**class** gpiozero.**MCP3004** (*channel=0*, *differential=False*, \*\**spi\_args*)

The **MCP3004**<sup>190</sup> 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), the MCP3002/3202 have 2 channels (0-1), and the MCP3001/3201/3301 only have 1 channel.

---

<sup>188</sup> <http://www.farnell.com/datasheets/630400.pdf>

<sup>189</sup> <http://www.farnell.com/datasheets/1599363.pdf>

<sup>190</sup> <http://www.farnell.com/datasheets/808965.pdf>

**differential**

If `True`, the device is operated in 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](#) (page 99) 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 certain devices operating in differential mode).

**class** `gpiozero.MCP3008` (*channel=0, differential=False, \*\*spi\_args*)

The [MCP3008](#)<sup>191</sup> 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), the MCP3002/3202 have 2 channels (0-1), and the MCP3001/3201/3301 only have 1 channel.

**differential**

If `True`, the device is operated in 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](#) (page 99) 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 certain devices operating in differential mode).

**class** `gpiozero.MCP3201` (*\*\*spi\_args*)

The [MCP3201](#)<sup>192</sup> is a 12-bit analog to digital converter with 1 channel. Please note that the MCP3201 always operates in differential mode, measuring the value of IN+ relative to IN-.

**value**

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

**class** `gpiozero.MCP3202` (*channel=0, differential=False, \*\*spi\_args*)

The [MCP3202](#)<sup>193</sup> 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), the MCP3002/3202 have 2 channels (0-1), and the MCP3001/3201/3301 only have 1 channel.

**differential**

If `True`, the device is operated in 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](#) (page 99) 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 certain devices operating in differential mode).

**class** `gpiozero.MCP3204` (*channel=0, differential=False, \*\*spi\_args*)

The [MCP3204](#)<sup>194</sup> is a 12-bit analog to digital converter with 4 channels (0-3).

<sup>191</sup> <http://www.farnell.com/datasheets/808965.pdf>

<sup>192</sup> <http://www.farnell.com/datasheets/1669366.pdf>

<sup>193</sup> <http://www.farnell.com/datasheets/1669376.pdf>

<sup>194</sup> <http://www.farnell.com/datasheets/808967.pdf>

**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), the MCP3002/3202 have 2 channels (0-1), and the MCP3001/3201/3301 only have 1 channel.

**differential**

If `True`, the device is operated in 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](#) (page 99) 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 certain devices operating in differential mode).

**class** `gpiozero.MCP3208` (*channel=0, differential=False, \*\*spi\_args*)

The [MCP3208](#)<sup>195</sup> 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), the MCP3002/3202 have 2 channels (0-1), and the MCP3001/3201/3301 only have 1 channel.

**differential**

If `True`, the device is operated in 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](#) (page 99) 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 certain devices operating in differential mode).

**class** `gpiozero.MCP3301` (*\*\*spi\_args*)

The [MCP3301](#)<sup>196</sup> is a signed 13-bit analog to digital converter. Please note that the MCP3301 always operates in differential mode measuring the difference between IN+ and IN-. Its 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](#)<sup>197</sup> 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), the MCP3002/3202 have 2 channels (0-1), and the MCP3001/3201/3301 only have 1 channel.

**differential**

If `True`, the device is operated in 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).

---

<sup>195</sup> <http://www.farnell.com/datasheets/808967.pdf>

<sup>196</sup> <http://www.farnell.com/datasheets/1669397.pdf>

<sup>197</sup> <http://www.farnell.com/datasheets/1486116.pdf>



Please refer to the device data-sheet to determine which channel is used as the relative base value (for example, when using an [MCP3304](#) (page 101) 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](#)<sup>198</sup> 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), the MCP3002/3202 have 2 channels (0-1), and the MCP3001/3201/3301 only have 1 channel.

**differential**

If `True`, the device is operated in 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 [MCP3304](#) (page 101) in differential mode, channel 0 is read relative to channel 1).

**value**

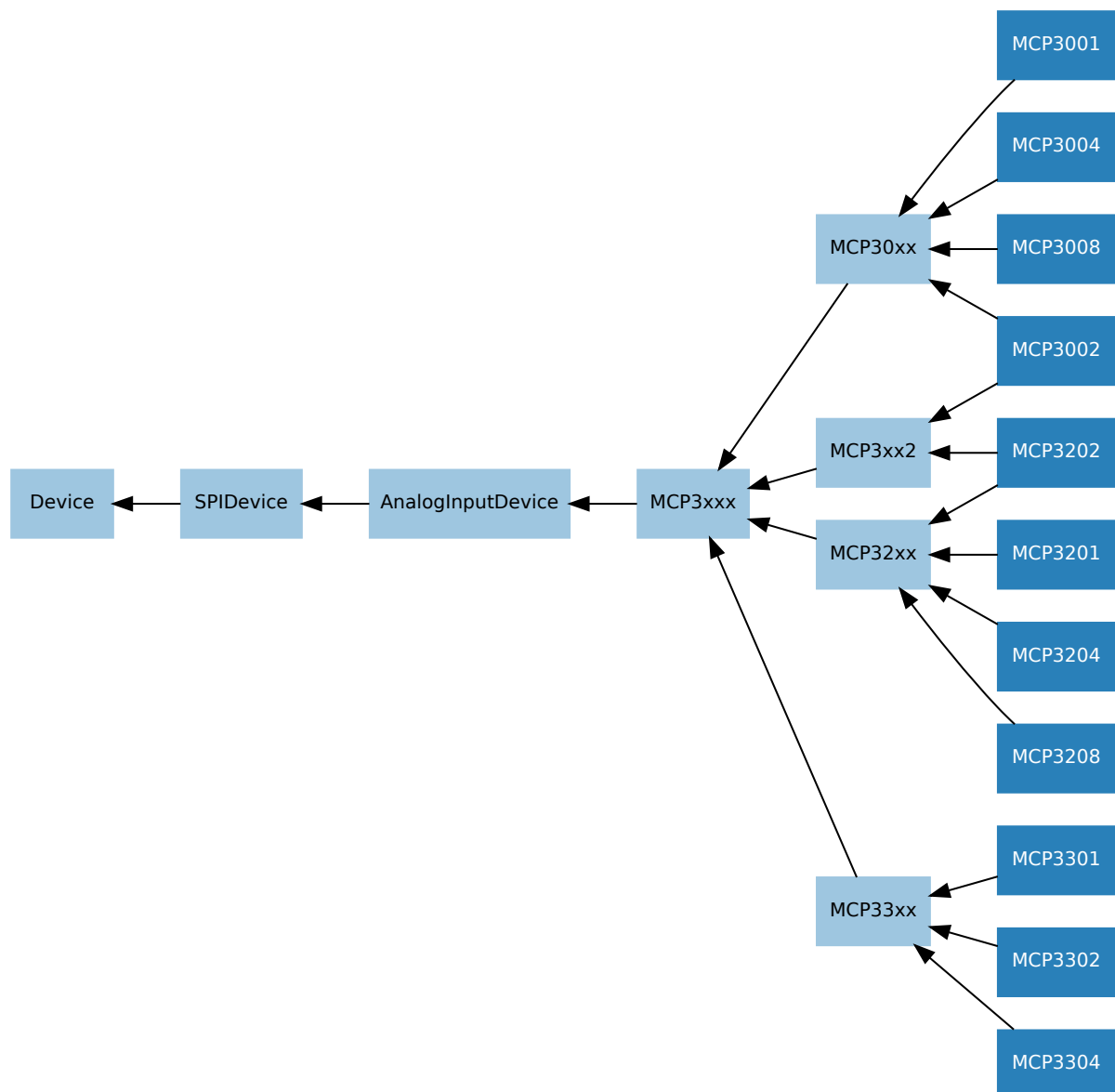
The current value read from the device, scaled to a value between 0 and 1 (or -1 to +1 for devices operating in differential mode).

## 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 (abstract classes are shaded lighter than concrete classes):

---

<sup>198</sup> <http://www.farnell.com/datasheets/1486116.pdf>



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

## AnalogInputDevice

**class** `gpiozero.AnalogInputDevice` (*bits*, *\*\*spi\_args*)

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

Typical analog input devices are [analog to digital converters](#)<sup>199</sup> (ADCs). Several classes are provided for specific ADC chips, including [MCP3004](#) (page 98), [MCP3008](#) (page 99), [MCP3204](#) (page 99), and [MCP3208](#) (page 100).

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)

```

<sup>199</sup> [https://en.wikipedia.org/wiki/Analog-to-digital\\_converter](https://en.wikipedia.org/wiki/Analog-to-digital_converter)

The *value* (page 103) 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* (page 82) 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 certain devices operating in differential mode).

## SPIDevice

**class** `gpiozero.SPIDevice` (*\*\*spi\_args*)

Extends *Device* (page 147). Represents a device that communicates via the SPI protocol.

See *SPI keyword args* (page 97) 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* (page 147) descendants can also be used as context managers using the `with`<sup>200</sup> statement. For example:

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

<sup>200</sup> [https://docs.python.org/3.5/reference/compound\\_stmts.html#with](https://docs.python.org/3.5/reference/compound_stmts.html#with)

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

---

## API - 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 *API - Input Devices* (page 69) and *API - Output Devices* (page 81) 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 *Basic Recipes* (page 3) page for more information.

---

### LEDBoard

**class** gpiozero.LEDBoard(\*pins, pwm=False, active\_high=True, initial\_value=False,  
pin\_factory=None, \*\*named\_pins)

Extends *LEDCollection* (page 137) 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*<sup>201</sup>) – 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* (page 105) instances to create trees of LEDs.
- **pwm** (*bool*<sup>202</sup>) – If True, construct *PWMLED* (page 82) instances for each pin. If False (the default), construct regular *LED* (page 81) instances. This parameter can only be specified as a keyword parameter.

---

<sup>201</sup> <https://docs.python.org/3.5/library/functions.html#int>

<sup>202</sup> <https://docs.python.org/3.5/library/functions.html#bool>

- **active\_high** (*bool*<sup>203</sup>) – If `True` (the default), the `on()` (page 107) method will set all the associated pins to HIGH. If `False`, the `on()` (page 107) method will set all pins to LOW (the `off()` (page 107) method always does the opposite). This parameter can only be specified as a keyword parameter.
- **initial\_value** (*bool*<sup>204</sup>) – 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.
- **pin\_factory** (*Factory* (page 166)) – See *API - Pins* (page 163) for more information (this is an advanced feature which most users can ignore).
- **\*\*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* (page 105) 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*<sup>205</sup>) – Number of seconds on. Defaults to 1 second.
- **off\_time** (*float*<sup>206</sup>) – Number of seconds off. Defaults to 1 second.
- **fade\_in\_time** (*float*<sup>207</sup>) – Number of seconds to spend fading in. Defaults to 0. Must be 0 if `pwm` was `False` when the class was constructed (*ValueError*<sup>208</sup> will be raised if not).
- **fade\_out\_time** (*float*<sup>209</sup>) – Number of seconds to spend fading out. Defaults to 0. Must be 0 if `pwm` was `False` when the class was constructed (*ValueError*<sup>210</sup> will be raised if not).
- **n** (*int*<sup>211</sup>) – Number of times to blink; `None` (the default) means forever.
- **background** (*bool*<sup>212</sup>) – 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:

---

<sup>203</sup> <https://docs.python.org/3.5/library/functions.html#bool>

<sup>204</sup> <https://docs.python.org/3.5/library/functions.html#bool>

<sup>205</sup> <https://docs.python.org/3.5/library/functions.html#float>

<sup>206</sup> <https://docs.python.org/3.5/library/functions.html#float>

<sup>207</sup> <https://docs.python.org/3.5/library/functions.html#float>

<sup>208</sup> <https://docs.python.org/3.5/library/exceptions.html#ValueError>

<sup>209</sup> <https://docs.python.org/3.5/library/functions.html#float>

<sup>210</sup> <https://docs.python.org/3.5/library/exceptions.html#ValueError>

<sup>211</sup> <https://docs.python.org/3.5/library/functions.html#int>

<sup>212</sup> <https://docs.python.org/3.5/library/functions.html#bool>

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

*Device* (page 147) descendants can also be used as context managers using the `with`<sup>213</sup> 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*<sup>214</sup>) – Number of seconds to spend fading in. Defaults to 1.
- **fade\_out\_time** (*float*<sup>215</sup>) – Number of seconds to spend fading out. Defaults to 1.
- **n** (*int*<sup>216</sup>) – Number of times to blink; None (the default) means forever.
- **background** (*bool*<sup>217</sup>) – 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* (page 107).

**source\_delay**

The delay (measured in seconds) in the loop used to read values from *source* (page 107). 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*.

<sup>213</sup> [https://docs.python.org/3.5/reference/compound\\_stmts.html#with](https://docs.python.org/3.5/reference/compound_stmts.html#with)

<sup>214</sup> <https://docs.python.org/3.5/library/functions.html#float>

<sup>215</sup> <https://docs.python.org/3.5/library/functions.html#float>

<sup>216</sup> <https://docs.python.org/3.5/library/functions.html#int>

<sup>217</sup> <https://docs.python.org/3.5/library/functions.html#bool>

## LEDBarGraph

**class** gpiozero.LEDBarGraph(\*pins, pwm=False, active\_high=True, initial\_value=0, pin\_factory=None)

Extends [LEDCollection](#) (page 137) 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](#) (page 109) and [values](#) (page 109) 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](#)<sup>218</sup>) – Specify the GPIO pins that the LEDs of the bar graph are attached to. You can designate as many pins as necessary.
- **pwm** ([bool](#)<sup>219</sup>) – If `True`, construct [PWMLLED](#) (page 82) instances for each pin. If `False` (the default), construct regular [LED](#) (page 81) instances. This parameter can only be specified as a keyword parameter.
- **active\_high** ([bool](#)<sup>220</sup>) – If `True` (the default), the [on\(\)](#) (page 108) method will set all the associated pins to HIGH. If `False`, the [on\(\)](#) (page 108) method will set all pins to LOW (the [off\(\)](#) (page 108) method always does the opposite). This parameter can only be specified as a keyword parameter.
- **initial\_value** ([float](#)<sup>221</sup>) – The initial [value](#) (page 109) 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.
- **pin\_factory** ([Factory](#) (page 166)) – See [API - Pins](#) (page 163) for more information (this is an advanced feature which most users can ignore).

**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.

<sup>218</sup> <https://docs.python.org/3.5/library/functions.html#int>

<sup>219</sup> <https://docs.python.org/3.5/library/functions.html#bool>

<sup>220</sup> <https://docs.python.org/3.5/library/functions.html#bool>

<sup>221</sup> <https://docs.python.org/3.5/library/functions.html#float>



**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* (page 109).

**source\_delay**

The delay (measured in seconds) in the loop used to read values from *source* (page 109). 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* (page 109) is effectively  $-1 < \text{value} \leq 1$ .

**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, pin_factory=None, **named_pins)
```

Extends *CompositeDevice* (page 138) and represents a generic button board or collection of buttons.

**Parameters**

- **\*pins** (*int*<sup>222</sup>) – Specify the GPIO pins that the buttons of the board are attached to. You can designate as many pins as necessary.
- **pull\_up** (*bool*<sup>223</sup>) – 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*<sup>224</sup>) – 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*<sup>225</sup>) – The length of time (in seconds) to wait after any button is pushed, until executing the *when\_held* (page 111) handler. Defaults to 1. This parameter can only be specified as a keyword parameter.

<sup>222</sup> <https://docs.python.org/3.5/library/functions.html#int>

<sup>223</sup> <https://docs.python.org/3.5/library/functions.html#bool>

<sup>224</sup> <https://docs.python.org/3.5/library/functions.html#float>

<sup>225</sup> <https://docs.python.org/3.5/library/functions.html#float>

- **hold\_repeat** (*bool*<sup>226</sup>) – If True, the *when\_held* (page 111) handler will be repeatedly executed as long as any buttons remain held, every *hold\_time* seconds. If False (the default) the *when\_held* (page 111) handler will be only be executed once per hold. This parameter can only be specified as a keyword parameter.
- **pin\_factory** (*Factory* (page 166)) – See *API - Pins* (page 163) for more information (this is an advanced feature which most users can ignore).
- **\*\*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*<sup>227</sup>) – 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*<sup>228</sup>) – 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*<sup>229</sup>) – 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*<sup>230</sup>) – 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* (page 111) event rather than when the device activated, in contrast to *active\_time* (page 149). If the device is not currently held, this is None.

**hold\_repeat**

If True, *when\_held* (page 111) will be executed repeatedly with *hold\_time* (page 110) seconds between each invocation.

**hold\_time**

The length of time (in seconds) to wait after the device is activated, until executing the *when\_held* (page 111) handler. If *hold\_repeat* (page 110) is True, this is also the length of time between invocations of *when\_held* (page 111).

**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* (page 110) seconds.

---

<sup>226</sup> <https://docs.python.org/3.5/library/functions.html#bool>

<sup>227</sup> <https://docs.python.org/3.5/library/functions.html#float>

<sup>228</sup> <https://docs.python.org/3.5/library/functions.html#float>

<sup>229</sup> <https://docs.python.org/3.5/library/functions.html#float>

<sup>230</sup> <https://docs.python.org/3.5/library/functions.html#float>

**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* (page 110) 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, pin_factory=None)
```

Extends *LEDBoard* (page 105) 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*<sup>231</sup>) – The GPIO pin that the red LED is attached to.
- **amber** (*int*<sup>232</sup>) – The GPIO pin that the amber LED is attached to.
- **green** (*int*<sup>233</sup>) – The GPIO pin that the green LED is attached to.
- **pwm** (*bool*<sup>234</sup>) – If `True`, construct *PWMLED* (page 82) instances to represent each LED. If `False` (the default), construct regular *LED* (page 81) instances.
- **initial\_value** (*bool*<sup>235</sup>) – 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*<sup>236</sup>) – 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`.
- **pin\_factory** (*Factory* (page 166)) – See *API - Pins* (page 163) for more information (this is an advanced feature which most users can ignore).

**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*<sup>237</sup>) – Number of seconds on. Defaults to 1 second.
- **off\_time** (*float*<sup>238</sup>) – Number of seconds off. Defaults to 1 second.
- **fade\_in\_time** (*float*<sup>239</sup>) – Number of seconds to spend fading in. Defaults to 0. Must be 0 if `pwm` was `False` when the class was constructed (*ValueError*<sup>240</sup> will be raised if not).
- **fade\_out\_time** (*float*<sup>241</sup>) – Number of seconds to spend fading out. Defaults to 0. Must be 0 if `pwm` was `False` when the class was constructed (*ValueError*<sup>242</sup> will be raised if not).
- **n** (*int*<sup>243</sup>) – Number of times to blink; `None` (the default) means forever.
- **background** (*bool*<sup>244</sup>) – 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.

<sup>231</sup> <https://docs.python.org/3.5/library/functions.html#int>

<sup>232</sup> <https://docs.python.org/3.5/library/functions.html#int>

<sup>233</sup> <https://docs.python.org/3.5/library/functions.html#int>

<sup>234</sup> <https://docs.python.org/3.5/library/functions.html#bool>

<sup>235</sup> <https://docs.python.org/3.5/library/functions.html#bool>

<sup>236</sup> <https://docs.python.org/3.5/library/functions.html#int>

<sup>237</sup> <https://docs.python.org/3.5/library/functions.html#float>

<sup>238</sup> <https://docs.python.org/3.5/library/functions.html#float>

<sup>239</sup> <https://docs.python.org/3.5/library/functions.html#float>

<sup>240</sup> <https://docs.python.org/3.5/library/exceptions.html#ValueError>

<sup>241</sup> <https://docs.python.org/3.5/library/functions.html#float>

<sup>242</sup> <https://docs.python.org/3.5/library/exceptions.html#ValueError>

<sup>243</sup> <https://docs.python.org/3.5/library/functions.html#int>

<sup>244</sup> <https://docs.python.org/3.5/library/functions.html#bool>

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* (page 147) descendents can also be used as context managers using the `with`<sup>245</sup> 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*<sup>246</sup>) – Number of seconds to spend fading in. Defaults to 1.
- **fade\_out\_time** (*float*<sup>247</sup>) – Number of seconds to spend fading out. Defaults to 1.
- **n** (*int*<sup>248</sup>) – Number of times to blink; None (the default) means forever.
- **background** (*bool*<sup>249</sup>) – 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* (page 114).

<sup>245</sup> [https://docs.python.org/3.5/reference/compound\\_stmts.html#with](https://docs.python.org/3.5/reference/compound_stmts.html#with)

<sup>246</sup> <https://docs.python.org/3.5/library/functions.html#float>

<sup>247</sup> <https://docs.python.org/3.5/library/functions.html#float>

<sup>248</sup> <https://docs.python.org/3.5/library/functions.html#int>

<sup>249</sup> <https://docs.python.org/3.5/library/functions.html#bool>

**source\_delay**

The delay (measured in seconds) in the loop used to read values from *source* (page 113). 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, *pin\_factory*=None)

Extends *RGBLED* (page 84) for the *PiBorg LedBorg*<sup>250</sup>: 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*<sup>251</sup>) – The initial color for the LedBorg. Defaults to black (0, 0, 0).
- **pwm** (*bool*<sup>252</sup>) – If True (the default), construct *PWMLED* (page 82) instances for each component of the LedBorg. If False, construct regular *LED* (page 81) instances, which prevents smooth color graduations.
- **pin\_factory** (*Factory* (page 166)) – See *API - Pins* (page 163) for more information (this is an advanced feature which most users can ignore).

**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*<sup>253</sup>) – Number of seconds on. Defaults to 1 second.
- **off\_time** (*float*<sup>254</sup>) – Number of seconds off. Defaults to 1 second.
- **fade\_in\_time** (*float*<sup>255</sup>) – Number of seconds to spend fading in. Defaults to 0. Must be 0 if *pwm* was False when the class was constructed (*ValueError*<sup>256</sup> will be raised if not).
- **fade\_out\_time** (*float*<sup>257</sup>) – Number of seconds to spend fading out. Defaults to 0. Must be 0 if *pwm* was False when the class was constructed (*ValueError*<sup>258</sup>

---

<sup>250</sup> <https://www.piborg.org/ledborg>

<sup>251</sup> <https://docs.python.org/3.5/library/stdtypes.html#tuple>

<sup>252</sup> <https://docs.python.org/3.5/library/functions.html#bool>

<sup>253</sup> <https://docs.python.org/3.5/library/functions.html#float>

<sup>254</sup> <https://docs.python.org/3.5/library/functions.html#float>

<sup>255</sup> <https://docs.python.org/3.5/library/functions.html#float>

<sup>256</sup> <https://docs.python.org/3.5/library/exceptions.html#ValueError>

<sup>257</sup> <https://docs.python.org/3.5/library/functions.html#float>

<sup>258</sup> <https://docs.python.org/3.5/library/exceptions.html#ValueError>



- **fade\_out\_time** (*float*<sup>265</sup>) – Number of seconds to spend fading out. Defaults to 1.
- **on\_color** (*tuple*<sup>266</sup>) – The color to use when the LED is “on”. Defaults to white.
- **off\_color** (*tuple*<sup>267</sup>) – The color to use when the LED is “off”. Defaults to black.
- **n** (*int*<sup>268</sup>) – Number of times to pulse; `None` (the default) means forever.
- **background** (*bool*<sup>269</sup>) – 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* (page 116) is `(0, 0, 0)`), this changes it to “fully” on (*value* (page 116) 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* (page 116).

**source\_delay**

The delay (measured in seconds) in the loop used to read values from *source* (page 116). 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*, *pin\_factory=None*)

Extends *LEDBoard* (page 105) for the *Ciseco Pi-LITER*<sup>270</sup>: 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:

---

<sup>265</sup> <https://docs.python.org/3.5/library/functions.html#float>

<sup>266</sup> <https://docs.python.org/3.5/library/stdtypes.html#tuple>

<sup>267</sup> <https://docs.python.org/3.5/library/stdtypes.html#tuple>

<sup>268</sup> <https://docs.python.org/3.5/library/functions.html#int>

<sup>269</sup> <https://docs.python.org/3.5/library/functions.html#bool>

<sup>270</sup> <http://shop.ciseco.co.uk/pi-liter-8-led-strip-for-the-raspberry-pi/>



```
from gpiozero import PiLiter

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

### Parameters

- **pwm** (*bool*<sup>271</sup>) – If `True`, construct `PWMLED` (page 82) instances for each pin. If `False` (the default), construct regular `LED` (page 81) instances.
- **initial\_value** (*bool*<sup>272</sup>) – 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.
- **pin\_factory** (`Factory` (page 166)) – See *API - Pins* (page 163) for more information (this is an advanced feature which most users can ignore).

**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*<sup>273</sup>) – Number of seconds on. Defaults to 1 second.
- **off\_time** (*float*<sup>274</sup>) – Number of seconds off. Defaults to 1 second.
- **fade\_in\_time** (*float*<sup>275</sup>) – Number of seconds to spend fading in. Defaults to 0. Must be 0 if `pwm` was `False` when the class was constructed (`ValueError`<sup>276</sup> will be raised if not).
- **fade\_out\_time** (*float*<sup>277</sup>) – Number of seconds to spend fading out. Defaults to 0. Must be 0 if `pwm` was `False` when the class was constructed (`ValueError`<sup>278</sup> will be raised if not).
- **n** (*int*<sup>279</sup>) – Number of times to blink; `None` (the default) means forever.
- **background** (*bool*<sup>280</sup>) – 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:

<sup>271</sup> <https://docs.python.org/3.5/library/functions.html#bool>

<sup>272</sup> <https://docs.python.org/3.5/library/functions.html#bool>

<sup>273</sup> <https://docs.python.org/3.5/library/functions.html#float>

<sup>274</sup> <https://docs.python.org/3.5/library/functions.html#float>

<sup>275</sup> <https://docs.python.org/3.5/library/functions.html#float>

<sup>276</sup> <https://docs.python.org/3.5/library/exceptions.html#ValueError>

<sup>277</sup> <https://docs.python.org/3.5/library/functions.html#float>

<sup>278</sup> <https://docs.python.org/3.5/library/exceptions.html#ValueError>

<sup>279</sup> <https://docs.python.org/3.5/library/functions.html#int>

<sup>280</sup> <https://docs.python.org/3.5/library/functions.html#bool>

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

*Device* (page 147) descendants can also be used as context managers using the `with`<sup>281</sup> 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*<sup>282</sup>) – Number of seconds to spend fading in. Defaults to 1.
- **fade\_out\_time** (*float*<sup>283</sup>) – Number of seconds to spend fading out. Defaults to 1.
- **n** (*int*<sup>284</sup>) – Number of times to blink; None (the default) means forever.
- **background** (*bool*<sup>285</sup>) – 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* (page 118).

**source\_delay**

The delay (measured in seconds) in the loop used to read values from *source* (page 118). 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*.

---

<sup>281</sup> [https://docs.python.org/3.5/reference/compound\\_stmts.html#with](https://docs.python.org/3.5/reference/compound_stmts.html#with)

<sup>282</sup> <https://docs.python.org/3.5/library/functions.html#float>

<sup>283</sup> <https://docs.python.org/3.5/library/functions.html#float>

<sup>284</sup> <https://docs.python.org/3.5/library/functions.html#int>

<sup>285</sup> <https://docs.python.org/3.5/library/functions.html#bool>

## PiLITEr Bar Graph

**class** gpiozero.**PiLiterBarGraph** (*pwm=False, initial\_value=0.0, pin\_factory=None*)

Extends [LEDBarGraph](#) (page 108) to treat the [Ciseco Pi-LITEr](#)<sup>286</sup> 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*<sup>287</sup>) – If `True`, construct [PWMLed](#) (page 82) instances for each pin. If `False` (the default), construct regular [LED](#) (page 81) instances.
- **initial\_value** (*float*<sup>288</sup>) – The initial *value* (page 119) of the graph given as a float between -1 and +1. Defaults to 0.0.
- **pin\_factory** ([Factory](#) (page 166)) – See [API - Pins](#) (page 163) for more information (this is an advanced feature which most users can ignore).

**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* (page 119).

**source\_delay**

The delay (measured in seconds) in the loop used to read values from *source* (page 119). 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* (page 119) is effectively -1 <

<sup>286</sup> <http://shop.ciseco.co.uk/pi-liter-8-led-strip-for-the-raspberry-pi/>

<sup>287</sup> <https://docs.python.org/3.5/library/functions.html#bool>

<sup>288</sup> <https://docs.python.org/3.5/library/functions.html#float>

value <= 1.

---

**values**

An infinite iterator of values read from *value*.

## PI-TRAFFIC

**class** gpiozero.**PiTraffic** (*pwm=False, initial\_value=False, pin\_factory=None*)

Extends *TrafficLights* (page 111) for the Low Voltage Labs PI-TRAFFIC<sup>289</sup> 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* (page 111).

**Parameters**

- **pwm** (*bool*<sup>290</sup>) – If *True*, construct *PWMLED* (page 82) instances to represent each LED. If *False* (the default), construct regular *LED* (page 81) instances.
- **initial\_value** (*bool*<sup>291</sup>) – 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.
- **pin\_factory** (*Factory* (page 166)) – See *API - Pins* (page 163) for more information (this is an advanced feature which most users can ignore).

**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*<sup>292</sup>) – Number of seconds on. Defaults to 1 second.
- **off\_time** (*float*<sup>293</sup>) – Number of seconds off. Defaults to 1 second.
- **fade\_in\_time** (*float*<sup>294</sup>) – Number of seconds to spend fading in. Defaults to 0. Must be 0 if *pwm* was *False* when the class was constructed (*ValueError*<sup>295</sup> will be raised if not).
- **fade\_out\_time** (*float*<sup>296</sup>) – Number of seconds to spend fading out. Defaults to 0. Must be 0 if *pwm* was *False* when the class was constructed (*ValueError*<sup>297</sup> will be raised if not).
- **n** (*int*<sup>298</sup>) – Number of times to blink; *None* (the default) means forever.
- **background** (*bool*<sup>299</sup>) – If *True*, start a background thread to continue blinking

---

<sup>289</sup> <http://lowvoltage labs.com/products/pi-traffic/>

<sup>290</sup> <https://docs.python.org/3.5/library/functions.html#bool>

<sup>291</sup> <https://docs.python.org/3.5/library/functions.html#bool>

<sup>292</sup> <https://docs.python.org/3.5/library/functions.html#float>

<sup>293</sup> <https://docs.python.org/3.5/library/functions.html#float>

<sup>294</sup> <https://docs.python.org/3.5/library/functions.html#float>

<sup>295</sup> <https://docs.python.org/3.5/library/exceptions.html#ValueError>

<sup>296</sup> <https://docs.python.org/3.5/library/functions.html#float>

<sup>297</sup> <https://docs.python.org/3.5/library/exceptions.html#ValueError>

<sup>298</sup> <https://docs.python.org/3.5/library/functions.html#int>

<sup>299</sup> <https://docs.python.org/3.5/library/functions.html#bool>



**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* (page 122).

**source\_delay**

The delay (measured in seconds) in the loop used to read values from *source* (page 122). 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*.

## Pi-Stop

**class** gpiozero.**PiStop** (*location=None, pwm=False, initial\_value=False, pin\_factory=None*)

Extends *TrafficLights* (page 111) for the *PiHardware Pi-Stop*<sup>305</sup>: a vertical traffic lights board.

The following example turns on the amber LED on a Pi-Stop connected to location A+:

```
from gpiozero import PiStop

traffic = PiStop('A+')
traffic.amber.on()
```

### Parameters

- **location** (*str*<sup>306</sup>) – The *location*<sup>307</sup> on the GPIO header to which the Pi-Stop is connected. Must be one of: A, A+, B, B+, C, D.
- **pwm** (*bool*<sup>308</sup>) – If *True*, construct *PWMLED* (page 82) instances to represent each LED. If *False* (the default), construct regular *LED* (page 81) instances.
- **initial\_value** (*bool*<sup>309</sup>) – 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.
- **pin\_factory** (*Factory* (page 166)) – See *API - Pins* (page 163) for more information (this is an advanced feature which most users can ignore).

**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*<sup>310</sup>) – Number of seconds on. Defaults to 1 second.
- **off\_time** (*float*<sup>311</sup>) – Number of seconds off. Defaults to 1 second.

---

<sup>305</sup> <https://pihw.wordpress.com/meltwaters-pi-hardware-kits/pi-stop/>

<sup>306</sup> <https://docs.python.org/3.5/library/stdtypes.html#str>

<sup>307</sup> [https://github.com/PiHw/Pi-Stop/blob/master/markdown\\_source/markdown/Discover-PiStop.md](https://github.com/PiHw/Pi-Stop/blob/master/markdown_source/markdown/Discover-PiStop.md)

<sup>308</sup> <https://docs.python.org/3.5/library/functions.html#bool>

<sup>309</sup> <https://docs.python.org/3.5/library/functions.html#bool>

<sup>310</sup> <https://docs.python.org/3.5/library/functions.html#float>

<sup>311</sup> <https://docs.python.org/3.5/library/functions.html#float>

- ```
close()
```

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

*Device* (page 147) descendants can also be used as context managers using the `with`<sup>318</sup> statement. For example:

**off** (*\*args*)

```
on ( *args)
```

**pulse** (*fade\_in\_time=1, fade\_out\_time=1, n=None, background=True*)

Make the device fade in and out repeatedly.

312 <https://docs.python.org/3.5/library/functions.html#float>  
313 <https://docs.python.org/3.5/library/exceptions.html#ValueError>  
314 <https://docs.python.org/3.5/library/functions.html#float>  
315 <https://docs.python.org/3.5/library/exceptions.html#ValueError>  
316 <https://docs.python.org/3.5/library/functions.html#int>  
317 <https://docs.python.org/3.5/library/functions.html#bool>  
318 [https://docs.python.org/3.5/reference/compound\\_stmts.html#with](https://docs.python.org/3.5/reference/compound_stmts.html#with)

- **fade\_in\_time** (*float*<sup>319</sup>) – Number of seconds to spend fading in. Defaults to 1.
- **fade\_out\_time** (*float*<sup>320</sup>) – Number of seconds to spend fading out. Defaults to 1.
- **n** (*int*<sup>321</sup>) – Number of times to blink; `None` (the default) means forever.
- **background** (*bool*<sup>322</sup>) – 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* (page 124).

**source\_delay**

The delay (measured in seconds) in the loop used to read values from *source* (page 124). 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, pin\_factory=None)

Extends *CompositeOutputDevice* (page 137) and is a generic class for HATs with traffic lights, a button and a buzzer.

### Parameters

- **lights** (*TrafficLights* (page 111)) – An instance of *TrafficLights* (page 111) representing the traffic lights of the HAT.
- **buzzer** (*Buzzer* (page 86)) – An instance of *Buzzer* (page 86) representing the buzzer on the HAT.
- **button** (*Button* (page 69)) – An instance of *Button* (page 69) representing the button on the HAT.
- **pin\_factory** (*Factory* (page 166)) – See *API - Pins* (page 163) for more information (this is an advanced feature which most users can ignore).

**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.

---

<sup>319</sup> <https://docs.python.org/3.5/library/functions.html#float>

<sup>320</sup> <https://docs.python.org/3.5/library/functions.html#float>

<sup>321</sup> <https://docs.python.org/3.5/library/functions.html#int>

<sup>322</sup> <https://docs.python.org/3.5/library/functions.html#bool>



**source**

The iterable to use as a source of values for *value* (page 125).

**source\_delay**

The delay (measured in seconds) in the loop used to read values from *source* (page 124). 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*, *pin\_factory=None*)

Extends *TrafficLightsBuzzer* (page 124) for the *Pi Supply FishDish*<sup>323</sup>: 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*<sup>324</sup>) – If *True*, construct *PWMLED* (page 82) instances to represent each LED. If *False* (the default), construct regular *LED* (page 81) instances.
- **pin\_factory** (*Factory* (page 166)) – See *API - Pins* (page 163) for more information (this is an advanced feature which most users can ignore).

**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* (page 125).

**source\_delay**

The delay (measured in seconds) in the loop used to read values from *source* (page 125). 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.

<sup>323</sup> <https://www.pi-supply.com/product/fish-dish-raspberry-pi-led-buzzer-board/>

<sup>324</sup> <https://docs.python.org/3.5/library/functions.html#bool>

**values**

An infinite iterator of values read from *value*.

## Traffic HAT

**class** `gpiozero.TrafficHat` (*pwm=False, pin\_factory=None*)

Extends [TrafficLightsBuzzer](#) (page 124) for the [Ryanteck Traffic HAT](#)<sup>325</sup>: 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*<sup>326</sup>) – If `True`, construct [PWMLed](#) (page 82) instances to represent each LED. If `False` (the default), construct regular [LED](#) (page 81) instances.
- **pin\_factory** ([Factory](#) (page 166)) – See [API - Pins](#) (page 163) for more information (this is an advanced feature which most users can ignore).

**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](#) (page 126).

**source\_delay**

The delay (measured in seconds) in the loop used to read values from [source](#) (page 126). 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, pin\_factory=None*)

Extends [CompositeDevice](#) (page 138) 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

---

<sup>325</sup> <https://ryanteck.uk/hats/1-traffic-hat-0635648607122.html>

<sup>326</sup> <https://docs.python.org/3.5/library/functions.html#bool>

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](#)<sup>327</sup>) – A tuple of two GPIO pins representing the forward and backward inputs of the left motor's controller.
- **right** ([tuple](#)<sup>328</sup>) – A tuple of two GPIO pins representing the forward and backward inputs of the right motor's controller.
- **pin\_factory** ([Factory](#) (page 166)) – See [API - Pins](#) (page 163) for more information (this is an advanced feature which most users can ignore).

**backward** (*speed=1*)

Drive the robot backward by running both motors backward.

**Parameters** **speed** ([float](#)<sup>329</sup>) – 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](#)<sup>330</sup>) – 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](#)<sup>331</sup>) – 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](#)<sup>332</sup>) – 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](#) (page 127).

**source\_delay**

The delay (measured in seconds) in the loop used to read values from [source](#) (page 127). Defaults to 0.01 seconds which is generally sufficient to keep CPU usage to a minimum while providing adequate responsiveness.

<sup>327</sup> <https://docs.python.org/3.5/library/stdtypes.html#tuple>

<sup>328</sup> <https://docs.python.org/3.5/library/stdtypes.html#tuple>

<sup>329</sup> <https://docs.python.org/3.5/library/functions.html#float>

<sup>330</sup> <https://docs.python.org/3.5/library/functions.html#float>

<sup>331</sup> <https://docs.python.org/3.5/library/functions.html#float>

<sup>332</sup> <https://docs.python.org/3.5/library/functions.html#float>

**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` (*pin\_factory=None*)

Extends *Robot* (page 126) for the Ryanteck motor controller board<sup>333</sup>.

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()
```

**Parameters** `pin_factory` (*Factory* (page 166)) – See *API - Pins* (page 163) for more information (this is an advanced feature which most users can ignore).

**backward** (*speed=1*)

Drive the robot backward by running both motors backward.

**Parameters** `speed` (*float*<sup>334</sup>) – 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*<sup>335</sup>) – 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*<sup>336</sup>) – 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*<sup>337</sup>) – 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* (page 129).

---

<sup>333</sup> <https://ryanteck.uk/add-ons/6-ryanteck-rpi-motor-controller-board-0635648607160.html>

<sup>334</sup> <https://docs.python.org/3.5/library/functions.html#float>

<sup>335</sup> <https://docs.python.org/3.5/library/functions.html#float>

<sup>336</sup> <https://docs.python.org/3.5/library/functions.html#float>

<sup>337</sup> <https://docs.python.org/3.5/library/functions.html#float>

**source\_delay**

The delay (measured in seconds) in the loop used to read values from *source* (page 128). 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 (*pin\_factory=None*)

Extends *Robot* (page 126) for the *CamJam #3 EduKit*<sup>338</sup> motor controller board.

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()
```

**Parameters** *pin\_factory* (*Factory* (page 166)) – See *API - Pins* (page 163) for more information (this is an advanced feature which most users can ignore).

**backward** (*speed=1*)

Drive the robot backward by running both motors backward.

**Parameters** *speed* (*float*<sup>339</sup>) – 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*<sup>340</sup>) – 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*<sup>341</sup>) – 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*<sup>342</sup>) – Speed at which to drive the motors, as a value between 0 (stopped) and 1 (full speed). The default is 1.

<sup>338</sup> [http://camjam.me/?page\\_id=1035](http://camjam.me/?page_id=1035)

<sup>339</sup> <https://docs.python.org/3.5/library/functions.html#float>

<sup>340</sup> <https://docs.python.org/3.5/library/functions.html#float>

<sup>341</sup> <https://docs.python.org/3.5/library/functions.html#float>

<sup>342</sup> <https://docs.python.org/3.5/library/functions.html#float>

**stop()**

Stop the robot.

**source**

The iterable to use as a source of values for *value* (page 130).

**source\_delay**

The delay (measured in seconds) in the loop used to read values from *source* (page 130). 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, pin\_factory=None*)

Extends *Device* (page 147) to represent an *Energenie socket*<sup>343</sup> 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*<sup>344</sup>) – Which socket this instance should control. This is an integer number between 1 and 4.
- **initial\_value** (*bool*<sup>345</sup>) – 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.
- **pin\_factory** (*Factory* (page 166)) – See *API - Pins* (page 163) for more information (this is an advanced feature which most users can ignore).

**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:

---

<sup>343</sup> <https://energenie4u.co.uk/index.php/catalogue/product/ENER002-2PI>

<sup>344</sup> <https://docs.python.org/3.5/library/functions.html#int>

<sup>345</sup> <https://docs.python.org/3.5/library/functions.html#bool>

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

*Device* (page 147) descendants can also be used as context managers using the `with`<sup>346</sup> 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* (page 131). 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*.

## StatusZero

**class** `gpiozero.StatusZero` (\*labels, *pwm=False*, *active\_high=True*, *initial\_value=False*, *pin\_factory=None*)

Extends *LEDBoard* (page 105) for The Pi Hut's *STATUS Zero*<sup>347</sup>: a Pi Zero sized add-on board with three sets of red/green LEDs to provide a status indicator.

The following example designates the first strip the label “wifi” and the second “raining”, and turns them green and red respectfully:

```
from gpiozero import StatusZero

status = StatusZero('wifi', 'raining')
status.wifi.green.on()
status.raining.red.on()
```

#### **Parameters**

- **\*labels** (*str*<sup>348</sup>) – Specify the names of the labels you wish to designate the strips to. You can list up to three labels. If no labels are given, three strips will be initialised with names ‘one’, ‘two’, and ‘three’. If some, but not all strips are given labels, any remaining strips will not be initialised.

<sup>346</sup> [https://docs.python.org/3.5/reference/compound\\_stmts.html#with](https://docs.python.org/3.5/reference/compound_stmts.html#with)

<sup>347</sup> <https://thepihut.com/statuszero>

<sup>348</sup> <https://docs.python.org/3.5/library/stdtypes.html#str>

- **pin\_factory** ([Factory](#) (page 166)) – See [API - Pins](#) (page 163) for more information (this is an advanced feature which most users can ignore).

**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](#)<sup>349</sup>) – Number of seconds on. Defaults to 1 second.
- **off\_time** ([float](#)<sup>350</sup>) – Number of seconds off. Defaults to 1 second.
- **fade\_in\_time** ([float](#)<sup>351</sup>) – Number of seconds to spend fading in. Defaults to 0. Must be 0 if `pwm` was `False` when the class was constructed ([ValueError](#)<sup>352</sup> will be raised if not).
- **fade\_out\_time** ([float](#)<sup>353</sup>) – Number of seconds to spend fading out. Defaults to 0. Must be 0 if `pwm` was `False` when the class was constructed ([ValueError](#)<sup>354</sup> will be raised if not).
- **n** ([int](#)<sup>355</sup>) – Number of times to blink; `None` (the default) means forever.
- **background** ([bool](#)<sup>356</sup>) – 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](#) (page 147) descendants can also be used as context managers using the `with`<sup>357</sup> statement. For example:

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

<sup>349</sup> <https://docs.python.org/3.5/library/functions.html#float>

<sup>350</sup> <https://docs.python.org/3.5/library/functions.html#float>

<sup>351</sup> <https://docs.python.org/3.5/library/functions.html#float>

<sup>352</sup> <https://docs.python.org/3.5/library/exceptions.html#ValueError>

<sup>353</sup> <https://docs.python.org/3.5/library/functions.html#float>

<sup>354</sup> <https://docs.python.org/3.5/library/exceptions.html#ValueError>

<sup>355</sup> <https://docs.python.org/3.5/library/functions.html#int>

<sup>356</sup> <https://docs.python.org/3.5/library/functions.html#bool>

<sup>357</sup> [https://docs.python.org/3.5/reference/compound\\_stmts.html#with](https://docs.python.org/3.5/reference/compound_stmts.html#with)



```
... 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*<sup>358</sup>) – Number of seconds to spend fading in. Defaults to 1.
- **fade\_out\_time** (*float*<sup>359</sup>) – Number of seconds to spend fading out. Defaults to 1.
- **n** (*int*<sup>360</sup>) – Number of times to blink; None (the default) means forever.
- **background** (*bool*<sup>361</sup>) – 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* (page 133).

**source\_delay**

The delay (measured in seconds) in the loop used to read values from *source* (page 133). 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*.

## StatusBoard

**class** gpiozero.**StatusBoard** (\*labels, pwm=False, active\_high=True, initial\_value=False, pin\_factory=None)

Extends *CompositeOutputDevice* (page 137) for The Pi Hut’s **STATUS**<sup>362</sup> board: a HAT sized add-on board with five sets of red/green LEDs and buttons to provide a status indicator with additional input.

The following example designates the first strip the label “wifi” and the second “raining”, turns the wifi green and then activates the button to toggle its lights when pressed:

<sup>358</sup> <https://docs.python.org/3.5/library/functions.html#float>

<sup>359</sup> <https://docs.python.org/3.5/library/functions.html#float>

<sup>360</sup> <https://docs.python.org/3.5/library/functions.html#int>

<sup>361</sup> <https://docs.python.org/3.5/library/functions.html#bool>

<sup>362</sup> <https://thepihut.com/status>

```
from gpiozero import StatusBoard

status = StatusBoard('wifi', 'raining')
status.wifi.lights.green.on()
status.wifi.button.when_pressed = status.wifi.lights.toggle
```

### Parameters

- **\*labels** (*str*<sup>363</sup>) – Specify the names of the labels you wish to designate the strips to. You can list up to five labels. If no labels are given, five strips will be initialised with names ‘one’ to ‘five’. If some, but not all strips are given labels, any remaining strips will not be initialised.
- **pin\_factory** (*Factory* (page 166)) – See *API - Pins* (page 163) for more information (this is an advanced feature which most users can ignore).

**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* (page 134).

**source\_delay**

The delay (measured in seconds) in the loop used to read values from *source* (page 134). 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*.

## SnowPi

**class** gpiozero.**SnowPi** (*pwm=False, initial\_value=False, pin\_factory=None*)

Extends *LEDBoard* (page 105) for the *Ryanteck SnowPi*<sup>364</sup> 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

---

<sup>363</sup> <https://docs.python.org/3.5/library/stdtypes.html#str>

<sup>364</sup> <https://ryanteck.uk/raspberry-pi/114-snowpi-the-gpio-snowman-for-raspberry-pi-0635648608303.html>

- **pwm** (*bool*<sup>365</sup>) – If `True`, construct *PWMLED* (page 82) instances to represent each LED. If `False` (the default), construct regular *LED* (page 81) instances.
- **initial\_value** (*bool*<sup>366</sup>) – 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.
- **pin\_factory** (*Factory* (page 166)) – See *API - Pins* (page 163) for more information (this is an advanced feature which most users can ignore).

**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*<sup>367</sup>) – Number of seconds on. Defaults to 1 second.
- **off\_time** (*float*<sup>368</sup>) – Number of seconds off. Defaults to 1 second.
- **fade\_in\_time** (*float*<sup>369</sup>) – Number of seconds to spend fading in. Defaults to 0. Must be 0 if `pwm` was `False` when the class was constructed (*ValueError*<sup>370</sup> will be raised if not).
- **fade\_out\_time** (*float*<sup>371</sup>) – Number of seconds to spend fading out. Defaults to 0. Must be 0 if `pwm` was `False` when the class was constructed (*ValueError*<sup>372</sup> will be raised if not).
- **n** (*int*<sup>373</sup>) – Number of times to blink; `None` (the default) means forever.
- **background** (*bool*<sup>374</sup>) – 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()
```

<sup>365</sup> <https://docs.python.org/3.5/library/functions.html#bool>

<sup>366</sup> <https://docs.python.org/3.5/library/functions.html#bool>

<sup>367</sup> <https://docs.python.org/3.5/library/functions.html#float>

<sup>368</sup> <https://docs.python.org/3.5/library/functions.html#float>

<sup>369</sup> <https://docs.python.org/3.5/library/functions.html#float>

<sup>370</sup> <https://docs.python.org/3.5/library/exceptions.html#ValueError>

<sup>371</sup> <https://docs.python.org/3.5/library/functions.html#float>

<sup>372</sup> <https://docs.python.org/3.5/library/exceptions.html#ValueError>

<sup>373</sup> <https://docs.python.org/3.5/library/functions.html#int>

<sup>374</sup> <https://docs.python.org/3.5/library/functions.html#bool>

*Device* (page 147) descendents can also be used as context managers using the `with`<sup>375</sup> 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*<sup>376</sup>) – Number of seconds to spend fading in. Defaults to 1.
- **fade\_out\_time** (*float*<sup>377</sup>) – Number of seconds to spend fading out. Defaults to 1.
- **n** (*int*<sup>378</sup>) – Number of times to blink; `None` (the default) means forever.
- **background** (*bool*<sup>379</sup>) – 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* (page 136).

**source\_delay**

The delay (measured in seconds) in the loop used to read values from *source* (page 136). 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:

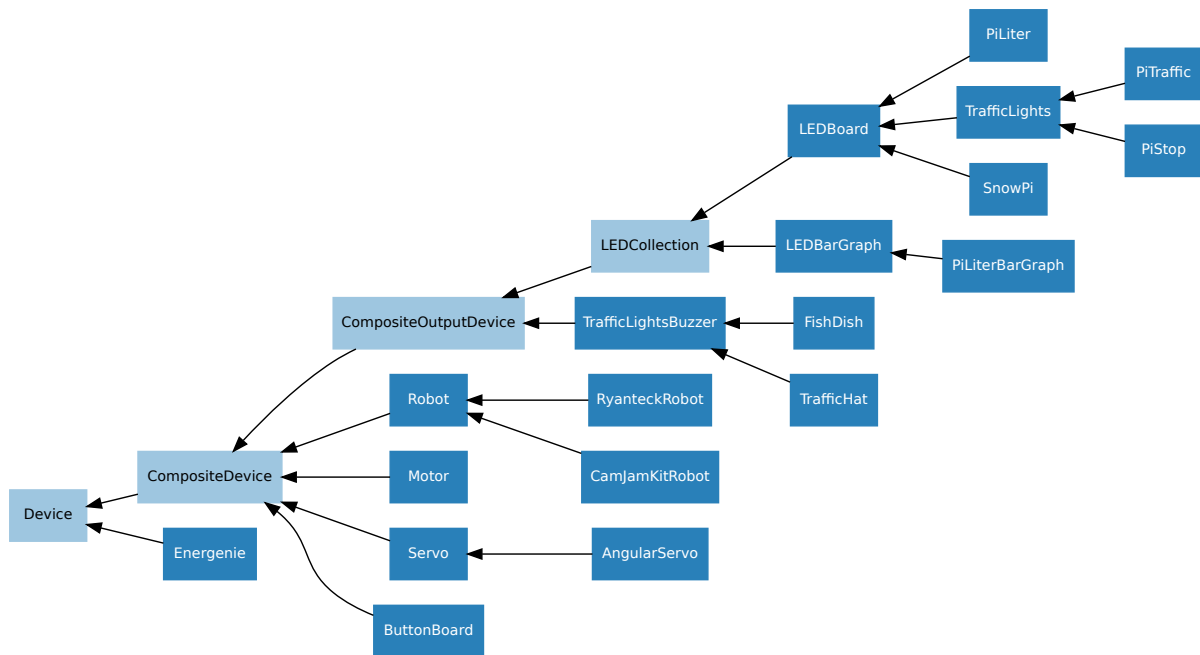
<sup>375</sup> [https://docs.python.org/3.5/reference/compound\\_stmts.html#with](https://docs.python.org/3.5/reference/compound_stmts.html#with)

<sup>376</sup> <https://docs.python.org/3.5/library/functions.html#float>

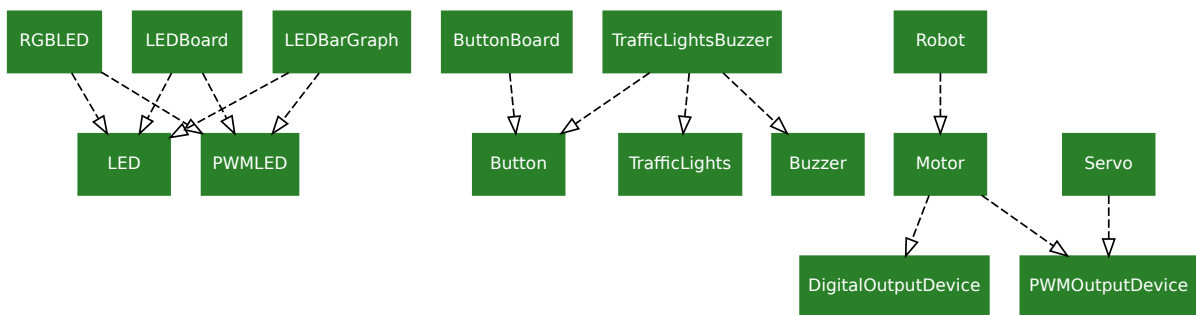
<sup>377</sup> <https://docs.python.org/3.5/library/functions.html#float>

<sup>378</sup> <https://docs.python.org/3.5/library/functions.html#int>

<sup>379</sup> <https://docs.python.org/3.5/library/functions.html#bool>



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** (\*pins, pwm=False, active\_high=True, initial\_value=False, pin\_factory=None, \*\*named\_pins)

Extends *CompositeOutputDevice* (page 137). Abstract base class for *LEDBoard* (page 105) and *LEDBarGraph* (page 108).

### leds

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

## CompositeOutputDevice

**class** gpiozero.**CompositeOutputDevice** (\*args, \_order=None, pin\_factory=None, \*\*kwargs)

Extends *CompositeDevice* (page 138) with *on()* (page 138), *off()* (page 138), and *toggle()* (page 138) methods for controlling subordinate output devices. Also extends *value* (page 138) to be writeable.

### Parameters

- **`_order`** (*list*<sup>380</sup>) – If specified, this is the order of named items specified by keyword arguments (to ensure that the *value* (page 138) 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.
- **`pin_factory`** (*Factory* (page 166)) – See *API - Pins* (page 163) for more information (this is an advanced feature which most users can ignore).

**`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*, *pin\_factory=None*, *\*\*kwargs*)

Extends *Device* (page 147). 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*<sup>381</sup>) – 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()
```

---

<sup>380</sup> <https://docs.python.org/3.5/library/stdtypes.html#list>

<sup>381</sup> <https://docs.python.org/3.5/library/stdtypes.html#list>

*Device* (page 147) descendents can also be used as context managers using the `with`<sup>382</sup> statement. For example:

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

```

---

<sup>382</sup> [https://docs.python.org/3.5/reference/compound\\_stmts.html#with](https://docs.python.org/3.5/reference/compound_stmts.html#with)





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* (page 143) 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`<sup>383</sup> instances.

The following example turns on a lamp attached to an *Energenie* (page 130) plug between 7 and 8 AM:

```
from gpiozero import TimeOfDay, Energenie
from datetime import time
from signal import pause

lamp = Energenie(0)
morning = TimeOfDay(time(7), time(8))

lamp.source = morning.values

pause()
```

#### Parameters

- **start\_time** (`time`<sup>384</sup>) – The time from which the device will be considered active.
- **end\_time** (`time`<sup>385</sup>) – The time after which the device will be considered inactive.

<sup>383</sup> <https://docs.python.org/3.5/library/datetime.html#datetime.time>

<sup>384</sup> <https://docs.python.org/3.5/library/datetime.html#datetime.time>

<sup>385</sup> <https://docs.python.org/3.5/library/datetime.html#datetime.time>

- **utc** (*bool*<sup>386</sup>) – 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* (page 143) 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* (page 148) to ensure the server is not flooded with pings):

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

google = PingServer('google.com')
led = LED(4)

led.source_delay = 60 # check once per minute
led.source = google.values

pause()
```

**Parameters** **host** (*str*<sup>387</sup>) – 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* (page 143) 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"
cpu = CPUTemperature(min_temp=50, max_temp=90)

print('Initial temperature: {}C'.format(cpu.temperature))

graph = LEDBarGraph(5, 6, 13, 19, 25, pwm=True)
graph.source = cpu.values

pause()
```

### Parameters

- **sensor\_file** (*str*<sup>388</sup>) – 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 millidegrees celsius.

---

<sup>386</sup> <https://docs.python.org/3.5/library/functions.html#bool>

<sup>387</sup> <https://docs.python.org/3.5/library/stdtypes.html#str>

<sup>388</sup> <https://docs.python.org/3.5/library/stdtypes.html#str>

- **min\_temp** (*float*<sup>389</sup>) – The temperature at which `value` will read 0.0. This defaults to 0.0.
- **max\_temp** (*float*<sup>390</sup>) – The temperature at which `value` will read 1.0. This defaults to 100.0.
- **threshold** (*float*<sup>391</sup>) – The temperature above which the device will be considered “active”. This defaults to 80.0.

**is\_active**

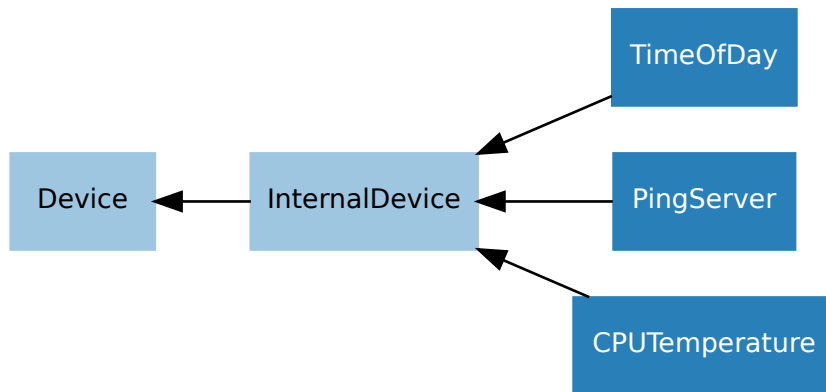
Returns `True` when the CPU *temperature* (page 143) exceeds the `threshold`.

**temperature**

Returns the current CPU temperature in degrees celsius.

## 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 (abstract classes are shaded lighter than concrete classes):



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* (page 147) 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.

<sup>389</sup> <https://docs.python.org/3.5/library/functions.html#float>

<sup>390</sup> <https://docs.python.org/3.5/library/functions.html#float>

<sup>391</sup> <https://docs.python.org/3.5/library/functions.html#float>



# CHAPTER 16

---

## API - Generic Classes

---

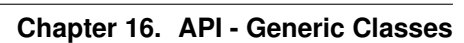
The GPIO Zero class hierarchy is quite extensive. It contains several base classes (most of which are documented in their corresponding chapters):

- *Device* (page 147) is the root of the hierarchy, implementing base functionality like `close()` (page 147) and context manager handlers.
- *GPIODevice* (page 80) represents individual devices that attach to a single GPIO pin
- *SPIDevice* (page 103) represents devices that communicate over an SPI interface (implemented as four GPIO pins)
- *InternalDevice* (page 143) represents devices that are entirely internal to the Pi (usually operating system related services)
- *CompositeDevice* (page 138) represents devices composed of multiple other devices like HATs

There are also several *mixin classes*<sup>392</sup> for adding important functionality at numerous points in the hierarchy, which is illustrated below (mixin classes are represented in purple, while abstract classes are shaded lighter):

---

<sup>392</sup> <https://en.wikipedia.org/wiki/Mixin>



## Device

**class** gpiozero.**Device** (\*, pin\_factory=None)

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 the *is\_active* (page 147) property, the *value* (page 147) property, and the *close()* (page 147) 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* (page 147) descendants can also be used as context managers using the *with*<sup>393</sup> 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()* (page 147) 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* (page 147). Unlike *value* (page 147), 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* (page 148) property to the class which returns an infinite generator of readings from the

<sup>393</sup> [https://docs.python.org/3.5/reference/compound\\_stmts.html#with](https://docs.python.org/3.5/reference/compound_stmts.html#with)

`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* (page 148) 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* (page 148). 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 (GPIONMeta) will use *\_shared\_key()* (page 148) 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()* (page 149) and *when\_deactivated()* (page 149) events to a device based on changes to the *is\_active* (page 147) property common to all devices. Also adds *wait\_for\_active()* (page 148) and *wait\_for\_inactive()* (page 149) 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*<sup>394</sup>) – 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*<sup>395</sup>) – 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* (page 148) to add the *when\_held* (page 149) event and the machinery to fire that event repeatedly (when *hold\_repeat* (page 149) is *True*) at intervals defined by *hold\_time* (page 149).

**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* (page 149) event rather than when the device activated, in contrast to *active\_time* (page 149). If the device is not currently held, this is *None*.

**hold\_repeat**

If *True*, *when\_held* (page 149) will be executed repeatedly with *hold\_time* (page 149) seconds between each invocation.

**hold\_time**

The length of time (in seconds) to wait after the device is activated, until executing the *when\_held* (page 149) handler. If *hold\_repeat* (page 149) is *True*, this is also the length of time between invocations of *when\_held* (page 149).

<sup>394</sup> <https://docs.python.org/3.5/library/functions.html#float>

<sup>395</sup> <https://docs.python.org/3.5/library/functions.html#float>

**is\_held**

When `True`, the device has been active for at least *hold\_time* (page 149) seconds.

**when\_held**

The function to run when the device has remained active for *hold\_time* (page 149) 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.

# CHAPTER 17

---

## API - Device Source Tools

---

GPIO Zero includes several utility routines which are intended to be used with the *Source/Values* (page 47) 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* (page 148) and *values* (page 148) deal with infinite iterators, another excellent source of utilities is the `itertools`<sup>396</sup> 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*

---

<sup>396</sup> <https://docs.python.org/3.5/library/itertools.html#module-itertools>

can optionally be used to add [hysteresis](https://en.wikipedia.org/wiki/Hysteresis)<sup>397</sup> 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*.

---

<sup>397</sup> <https://en.wikipedia.org/wiki/Hysteresis>

`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()` (page 153), *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<sup>398</sup> 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()
```

---

<sup>398</sup> [https://en.wikipedia.org/wiki/Logical\\_conjunction](https://en.wikipedia.org/wiki/Logical_conjunction)

`gpiozero.tools.any_values(*values)`

Returns the [logical disjunction](https://en.wikipedia.org/wiki/Logical_disjunction)<sup>399</sup> 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)
```

<sup>399</sup> [https://en.wikipedia.org/wiki/Logical\\_disjunction](https://en.wikipedia.org/wiki/Logical_disjunction)

```
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.alternating_values` (*initial\_value=False*)

Provides an infinite source of values alternating between True and False, starting with *initial\_value* (which defaults to False). For example, to produce a flashing LED:

```
from gpiozero import LED
from gpiozero.tools import alternating_values
from signal import pause

red = LED(2)

red.source_delay = 0.5
red.source = alternating_values()

pause()
```

`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()` (page 154).

`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()` (page 154).



`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()` (page 154).



## API - Pi Information

The GPIO Zero library also contains a database of information about the various revisions of the Raspberry Pi computer. 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* (page 159) instance containing information about a *revision* of the Raspberry Pi.

**Parameters** `revision (str400)` – 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()`<sup>401</sup> 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. The object can be used in format strings with various custom format specifications:

```
from gpiozero import *

print('{0}'.format(pi_info()))
print('{0:full}'.format(pi_info()))
print('{0:board}'.format(pi_info()))
print('{0:specs}'.format(pi_info()))
print('{0:headers}'.format(pi_info()))
```

'color' and 'mono' can be prefixed to format specifications to force the use of ANSI color codes<sup>402</sup>. If neither is specified, ANSI codes will only be used if stdout is detected to be a tty:

```
print('{0:color board}'.format(pi_info())) # force use of ANSI codes
print('{0:mono board}'.format(pi_info())) # force plain ASCII
```

**physical\_pin (function)**

Return the physical pin supporting the specified *function*. If no pins support the desired *function*, this function raises *PinNoPins* (page 179). If multiple pins support the desired *function*,

<sup>400</sup> <https://docs.python.org/3.5/library/stdtypes.html#str>

<sup>401</sup> <https://docs.python.org/3.5/library/collections.html#collections.namedtuple>

<sup>402</sup> [https://en.wikipedia.org/wiki/ANSI\\_escape\\_code](https://en.wikipedia.org/wiki/ANSI_escape_code)

*PinMultiplePins* (page 179) will be raised (use *physical\_pins()* (page 160) if you expect multiple pins in the result, such as for electrical ground).

**Parameters** *function* (*str*<sup>403</sup>) – 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*<sup>404</sup> which is not indexable. Use *physical\_pin()* (page 159) if you are expecting a single return value.

**Parameters** *function* (*str*<sup>405</sup>) – 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.

**pprint** (*color=None*)

Pretty-print a representation of the board along with header diagrams.

If *color* is *None* (the default), the diagram will include ANSI color codes if stdout is a color-capable terminal. Otherwise *color* can be set to *True* or *False* to force color or monochrome output.

**pulled\_up** (*function*)

Returns a bool indicating whether a physical pull-up is attached to the pin supporting the specified *function*. Either *PinNoPins* (page 179) or *PinMultiplePins* (page 179) may be raised if the function is not associated with a single pin.

**Parameters** *function* (*str*<sup>406</sup>) – 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*<sup>407</sup>) 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.

---

<sup>403</sup> <https://docs.python.org/3.5/library/stdtypes.html#str>

<sup>404</sup> <https://docs.python.org/3.5/library/stdtypes.html#set>

<sup>405</sup> <https://docs.python.org/3.5/library/stdtypes.html#str>

<sup>406</sup> <https://docs.python.org/3.5/library/stdtypes.html#str>

<sup>407</sup> [https://en.wikipedia.org/wiki/System\\_on\\_a\\_chip](https://en.wikipedia.org/wiki/System_on_a_chip)

---

**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 *HeaderInfo* (page 161) tuples. For example, to obtain information about header P1 you would query `headers['P1']`. To obtain information about pin 12 on header J8 you would query `headers['J8'].pins[12]`.

A rendered version of this data can be obtained by using the *PiBoardInfo* (page 159) object in a format string:

```
from gpiozero import *
print('{0:headers}'.format(pi_info()))
```

**board**

An ASCII art rendition of the board, primarily intended for console pretty-print usage. A more usefully rendered version of this data can be obtained by using the *PiBoardInfo* (page 159) object in a format string. For example:

```
from gpiozero import *
print('{0:board}'.format(pi_info()))
```

**class gpiozero.HeaderInfo**

This class is a `namedtuple()`<sup>408</sup> derivative used to represent information about a pin header on a board. The object can be used in a format string with various custom specifications:

```
from gpiozero import *

print('{0}'.format(pi_info().headers['J8']))
print('{0:full}'.format(pi_info().headers['J8']))
print('{0:col2}'.format(pi_info().headers['P1']))
print('{0:row1}'.format(pi_info().headers['P1']))
```

---

<sup>408</sup> <https://docs.python.org/3.5/library/collections.html#collections.namedtuple>

`'color'` and `'mono'` can be prefixed to format specifications to force the use of ANSI color codes<sup>409</sup>. If neither is specified, ANSI codes will only be used if stdout is detected to be a tty:

```
print('{0:color row2}'.format(pi_info().headers['J8'])) # force use of ANSI_
↪codes
print('{0:mono row2}'.format(pi_info().headers['P1'])) # force plain ASCII
```

The following attributes are defined:

**pprint** (*color=None*)

Pretty-print a diagram of the header pins.

If *color* is *None* (the default, the diagram will include ANSI color codes if stdout is a color-capable terminal). Otherwise *color* can be set to *True* or *False* to force color or monochrome output.

**name**

The name of the header, typically as it appears silk-screened on the board (e.g. “P1” or “J8”).

**rows**

The number of rows on the header.

**columns**

The number of columns on the header.

**pins**

A dictionary mapping physical pin numbers to *PinInfo* (page 162) tuples.

**class** `gpiozero.PinInfo`

This class is a `namedtuple()`<sup>410</sup> 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.

**row**

An integer indicating on which row the pin is physically located in the header (1-based)

**col**

An integer indicating in which column the pin is physically located in the header (1-based)

---

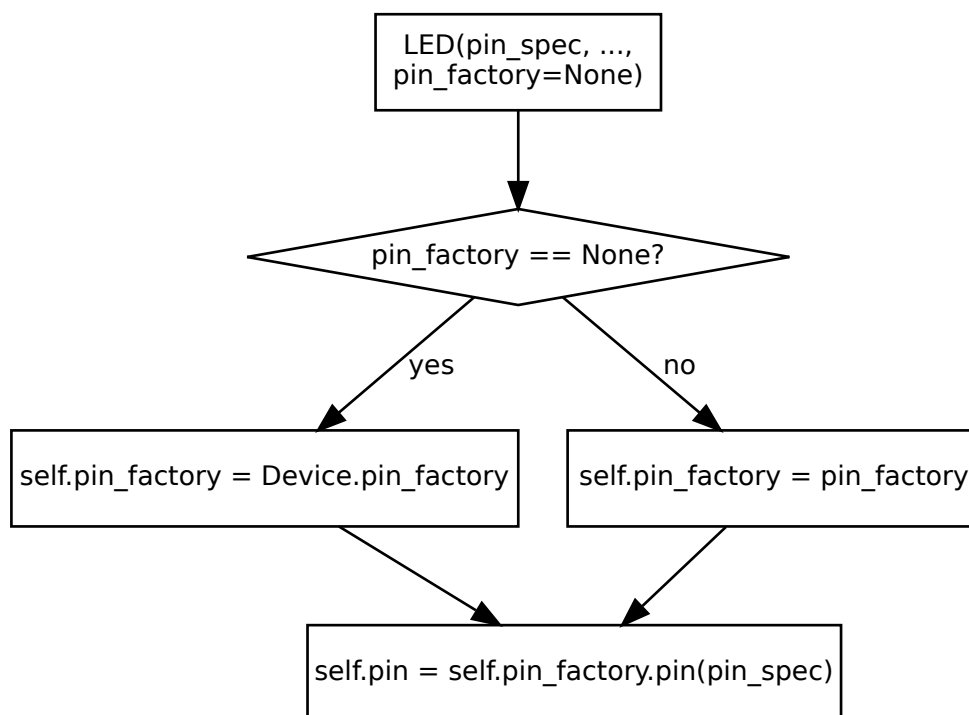
<sup>409</sup> [https://en.wikipedia.org/wiki/ANSI\\_escape\\_code](https://en.wikipedia.org/wiki/ANSI_escape_code)

<sup>410</sup> <https://docs.python.org/3.5/library/collections.html#collections.namedtuple>

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 pin specification. This is passed to a pin *Factory* (page 166) which turns it into a *Pin* (page 167) implementation. The default factory can be queried (and changed) with `Device.pin_factory`, i.e. the `pin_factory` attribute of the *Device* (page 147) class. However, all classes accept a `pin_factory` keyword argument to their constructors permitting the factory to be overridden on a per-device basis (the reason for allowing per-device factories is made apparent later in the *Configuring Remote GPIO* (page 35) chapter).

This is illustrated in the following flow-chart:



The default factory is constructed when GPIO Zero is first imported; if no default factory can be constructed (e.g. because no GPIO implementations are installed, or all of them fail to load for whatever reason), an `ImportError`<sup>411</sup> will be raised.

## Changing the pin factory

The default pin factory can be replaced by specifying a value for the `GPIOZERO_PIN_FACTORY` environment variable. For example:

```
$ GPIOZERO_PIN_FACTORY=native python
Python 3.4.2 (default, Oct 19 2014, 13:31:11)
[GCC 4.9.1] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import gpiozero
>>> gpiozero.Device.pin_factory
<gpiozero.pins.native.NativeFactory object at 0x762c26b0>
```

To set the `GPIOZERO_PIN_FACTORY` for the rest of your session you can export this value:

```
$ export GPIOZERO_PIN_FACTORY=native
$ python
Python 3.4.2 (default, Oct 19 2014, 13:31:11)
[GCC 4.9.1] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import gpiozero
>>> gpiozero.Device.pin_factory
<gpiozero.pins.native.NativeFactory object at 0x762c26b0>
>>> quit()
$ python
Python 3.4.2 (default, Oct 19 2014, 13:31:11)
[GCC 4.9.1] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import gpiozero
>>> gpiozero.Device.pin_factory
<gpiozero.pins.native.NativeFactory object at 0x76401330>
```

If you add the `export` command to your `~/ .bashrc` file, you'll set the default pin factory for all future sessions too.

The following values, and the corresponding *Factory* (page 166) and *Pin* (page 167) classes are listed in the table below. Factories are listed in the order that they are tried by default.

| Name     | Factory class                                                   | Pin class                                                  |
|----------|-----------------------------------------------------------------|------------------------------------------------------------|
| rpig-pio | <code>gpiozero.pins.rpigpio.RPiGPIOFactory</code><br>(page 173) | <code>gpiozero.pins.rpigpio.RPiGIOPin</code><br>(page 173) |
| rpio     | <code>gpiozero.pins.rpio.RPIOFactory</code><br>(page 174)       | <code>gpiozero.pins.rpio.RPIOPin</code><br>(page 174)      |
| pig-pio  | <code>gpiozero.pins.pigpio.PiGPIOFactory</code><br>(page 174)   | <code>gpiozero.pins.pigpio.PiGIOPin</code><br>(page 175)   |
| na-tive  | <code>gpiozero.pins.native.NativeFactory</code><br>(page 175)   | <code>gpiozero.pins.native.NativePin</code><br>(page 175)  |

If you need to change the default pin factory from within a script, either set `Device.pin_factory` to the new factory instance to use:

```
from gpiozero.pins.native import NativeFactory
from gpiozero import Device, LED
```

<sup>411</sup> <https://docs.python.org/3.5/library/exceptions.html#ImportError>



```
Device.pin_factory = NativeFactory()

# These will now implicitly use NativePin instead of
# RPiGPIOPin
led1 = LED(16)
led2 = LED(17)
```

Or use the `pin_factory` keyword parameter mentioned above:

```
from gpiozero.pins.native import NativeFactory
from gpiozero import LED

my_factory = NativeFactory()

# This will use NativePin instead of RPiGPIOPin for led1
# but led2 will continue to use RPiGPIOPin
led1 = LED(16, pin_factory=my_factory)
led2 = LED(17)
```

Certain factories may take default information from additional sources. For example, to default to creating pins with `gpiozero.pins.pigpio.PiGPIOPin` (page 175) on a remote pi called `remote-pi` you can set the `PIGPIO_ADDR` environment variable when running your script:

```
$ GPIOZERO_PIN_FACTORY=pigpio PIGPIO_ADDR=remote-pi python3 my_script.py
```

Like the `GPIOZERO_PIN_FACTORY` value, these can be exported from your `~/ .bashrc` script too.

**Warning:** The astute and mischievous reader may note that it is possible to mix factories, e.g. using `RPiGPIOFactory` for one pin, and `NativeFactory` 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.

Sensible uses of multiple pin factories are given in *Configuring Remote GPIO* (page 35).

## Mock pins

There's also a `gpiozero.pins.mock.MockFactory` (page 175) which generates entirely fake pins. This was originally intended for GPIO Zero developers who wish to write tests for devices without having to have the physical device wired in to their Pi. However, they have also proven relatively useful in developing GPIO Zero scripts without having a Pi to hand. This pin factory will never be loaded by default; it must be explicitly specified. For example:

```
from gpiozero.pins.mock import MockFactory
from gpiozero import Device, Button, LED
from time import sleep

# Set the default pin factory to a mock factory
Device.pin_factory = MockFactory()

# Construct a couple of devices attached to mock pins 16 and 17, and link the
# devices
led = LED(17)
btn = Button(16)
led.source = btn.values

# Here the button isn't "pushed" so the LED's value should be False
print(led.value)
```

```
# Get a reference to mock pin 16 (used by the button)
btn_pin = Device.pin_factory.pin(16)

# Drive the pin low (this is what would happen eletrically when the button is
# pushed)
btn_pin.drive_low()
sleep(0.1) # give source some time to re-read the button state
print(led.value)

btn_pin.drive_high()
sleep(0.1)
print(led.value)
```

Several sub-classes of mock pins exist for emulating various other things (pins that do/don't support PWM, pins that are connected together, pins that drive high after a delay, etc). Interested users are invited to read the GPIO Zero test suite for further examples of usage.

## Base classes

### **class** `gpiozero.Factory`

Generates pins and SPI interfaces for devices. This is an abstract base class for pin factories. Descendents *may* override the following methods, if applicable:

- `close()` (page 166)
- `reserve_pins()` (page 166)
- `release_pins()` (page 166)
- `release_all()` (page 166)
- `pin()` (page 166)
- `spi()` (page 166)
- `_get_pi_info()`

#### **close()**

Closes the pin factory. This is expected to clean up all resources manipulated by the factory. It is typically called at script termination.

#### **pin** (*spec*)

Creates an instance of a [Pin](#) (page 167) descendent representing the specified pin.

**Warning:** Descendents must ensure that pin instances representing the same hardware are identical; i.e. two separate invocations of `pin()` (page 166) for the same pin specification must return the same object.

#### **release\_all** (*reserver*)

Releases all pin reservations taken out by *reserver*. See [release\\_pins\(\)](#) (page 166) for further information).

#### **release\_pins** (*reserver*, *\*pins*)

Releases the reservation of *reserver* against *pins*. This is typically called during `Device.close()` (page 147) to clean up reservations taken during construction. Releasing a reservation that is not currently held will be silently ignored (to permit clean-up after failed / partial construction).

#### **reserve\_pins** (*requester*, *\*pins*)

Called to indicate that the device reserves the right to use the specified *pins*. This should be done during device construction. If pins are reserved, you must ensure that the reservation is released by eventually called [release\\_pins\(\)](#) (page 166).

**spi** (*\*\*spi\_args*)

Returns an instance of an *SPI* (page 169) interface, for the specified *SPI port* and *device*, or for the specified pins (*clock\_pin*, *mosi\_pin*, *miso\_pin*, and *select\_pin*). Only one of the schemes can be used; attempting to mix *port* and *device* with pin numbers will raise *SPIBadArgs* (page 178).

**pi\_info**

Returns a *PiBoardInfo* (page 159) instance representing the Pi that instances generated by this factory 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*.

**class** gpiozero.**Pin**

Abstract base class representing a pin attached to some form of controller, be it GPIO, SPI, ADC, etc.

Descendents should override property getters and setters to accurately represent the capabilities of pins.

Descendents *must* override the following methods:

- `_get_function()`
- `_set_function()`
- `_get_state()`

Descendents *may* additionally override the following methods, if applicable:

- `close()` (page 167)
- `output_with_state()` (page 167)
- `input_with_pull()` (page 167)
- `_set_state()`
- `_get_frequency()`
- `_set_frequency()`
- `_get_pull()`
- `_set_pull()`
- `_get_bounce()`
- `_set_bounce()`
- `_get_edges()`
- `_set_edges()`
- `_get_when_changed()`
- `_set_when_changed()`

**close** ()

Cleans up the resources allocated to the pin. After this method is called, this *Pin* (page 167) 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:

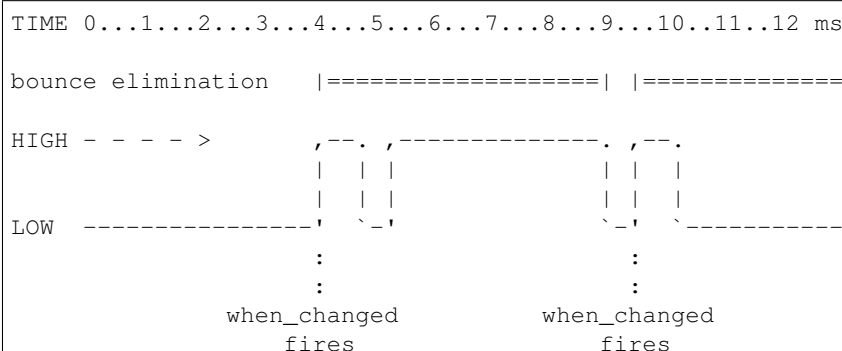
```
pin.function = 'output'
pin.state = state
```

However, descendants 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).

**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`.

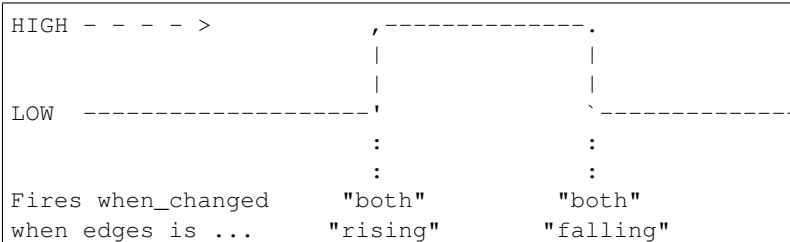
For example, if [edges](#) (page 168) is currently “rising”, [bounce](#) (page 168) is currently 5/1000 (5ms), then the waveform below will only fire [when\\_changed](#) (page 169) on two occasions despite there being three rising edges:



If the pin does not support edge detection, attempts to set this property will raise [PinEdgeDetectUnsupported](#) (page 179). 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](#) (page 169). 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](#) (page 179).

**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](#) (page 179) 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* (page 179) 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* (page 179). If the specified value is not supported by the underlying hardware, *PinInvalidPull* (page 179) 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* (page 169) for more information):

|      |           |   |       |
|------|-----------|---|-------|
| HIGH | - - - - > | , | ----- |
|      |           |   |       |
| LOW  | -----     |   |       |

Descendents which implement analog, or analog-like capabilities can return values between 0 and 1. For example, pins implementing PWM (where *frequency* (page 168) is not *None*) return a value between 0.0 and 1.0 representing the current PWM duty cycle.

If a pin is currently configured for input, and an attempt is made to set this attribute, *PinSetInput* (page 179) will be raised. If an invalid value is specified for this attribute, *PinInvalidState* (page 179) 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* (page 168) 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* (page 179).

#### **class gpiozero.SPI**

Abstract interface for *Serial Peripheral Interface*<sup>412</sup> (SPI) implementations. Descendents *must* override the following methods:

- *transfer()* (page 170)
- *\_get\_clock\_mode()*

Descendents *may* override the following methods, if applicable:

- *read()* (page 169)
- *write()* (page 170)
- *\_set\_clock\_mode()*
- *\_get\_lsb\_first()*
- *\_set\_lsb\_first()*
- *\_get\_select\_high()*
- *\_set\_select\_high()*
- *\_get\_bits\_per\_word()*
- *\_set\_bits\_per\_word()*

<sup>412</sup> [https://en.wikipedia.org/wiki/Serial\\_Peripheral\\_Interface\\_Bus](https://en.wikipedia.org/wiki/Serial_Peripheral_Interface_Bus)

**read(*n*)**

Read *n* words of data from the SPI interface, returning them as a sequence of unsigned ints, each no larger than the configured `bits_per_word` (page 170) of the interface.

This method is typically used with read-only devices that feature half-duplex communication. See `transfer()` (page 170) for full duplex communication.

**transfer(*data*)**

Write *data* to the SPI interface. *data* must be a sequence of unsigned integer words each of which will fit within the configured `bits_per_word` (page 170) of the interface. The method returns the sequence of words read from the interface while writing occurred (full duplex communication).

The length of the sequence returned dictates the number of words of *data* written to the interface. Each word in the returned sequence will be an unsigned integer no larger than the configured `bits_per_word` (page 170) of the interface.

**write(*data*)**

Write *data* to the SPI interface. *data* must be a sequence of unsigned integer words each of which will fit within the configured `bits_per_word` (page 170) of the interface. The method returns the number of words written to the interface (which may be less than or equal to the length of *data*).

This method is typically used with write-only devices that feature half-duplex communication. See `transfer()` (page 170) for full duplex communication.

**bits\_per\_word**

Controls the number of bits that make up a word, and thus where the word boundaries appear in the data stream, and the maximum value of a word. Defaults to 8 meaning that words are effectively bytes.

Several implementations do not support non-byte-sized words.

**clock\_mode**

Presents a value representing the `clock_polarity` (page 171) and `clock_phase` (page 170) attributes combined according to the following table:

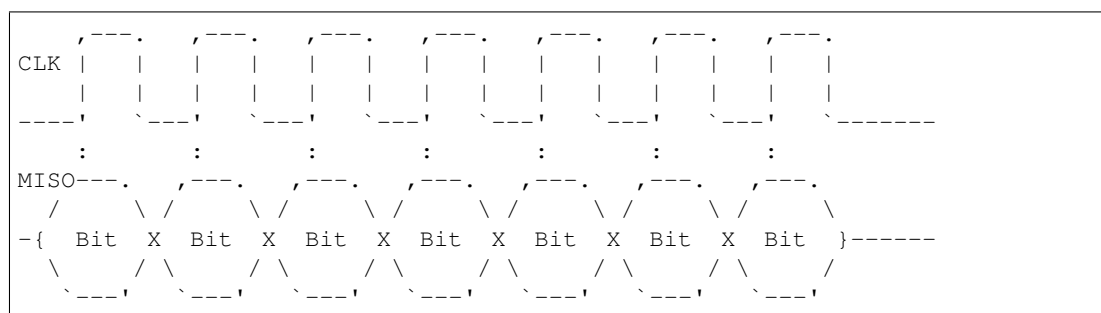
| mode | polarity (CPOL) | phase (CPHA) |
|------|-----------------|--------------|
| 0    | False           | False        |
| 1    | False           | True         |
| 2    | True            | False        |
| 3    | True            | True         |

Adjusting this value adjusts both the `clock_polarity` (page 171) and `clock_phase` (page 170) attributes simultaneously.

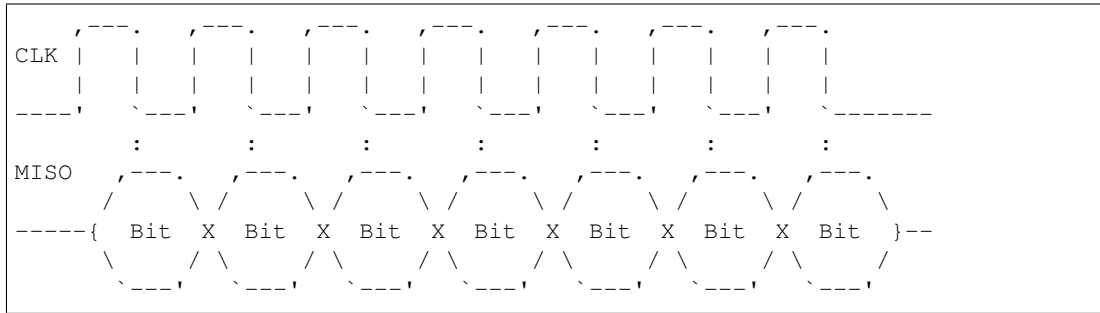
**clock\_phase**

The phase of the SPI clock pin. If this is `False` (the default), data will be read from the MISO pin when the clock pin activates. Setting this to `True` will cause data to be read from the MISO pin when the clock pin deactivates. On many data sheets this is documented as the CPHA value. Whether the clock edge is rising or falling when the clock is considered activated is controlled by the `clock_polarity` (page 171) attribute (corresponding to CPOL).

The following diagram indicates when data is read when `clock_polarity` (page 171) is `False`, and `clock_phase` (page 170) is `False` (the default), equivalent to CPHA 0:



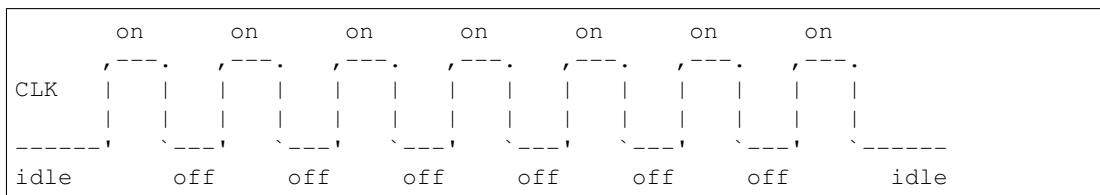
The following diagram indicates when data is read when `clock_polarity` (page 171) is False, but `clock_phase` (page 170) is True, equivalent to CPHA 1:



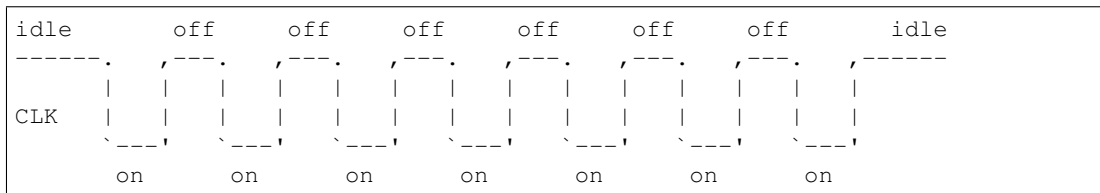
### `clock_polarity`

The polarity of the SPI clock pin. If this is False (the default), the clock pin will idle low, and pulse high. Setting this to True will cause the clock pin to idle high, and pulse low. On many data sheets this is documented as the CPOL value.

The following diagram illustrates the waveform when `clock_polarity` (page 171) is False (the default), equivalent to CPOL 0:



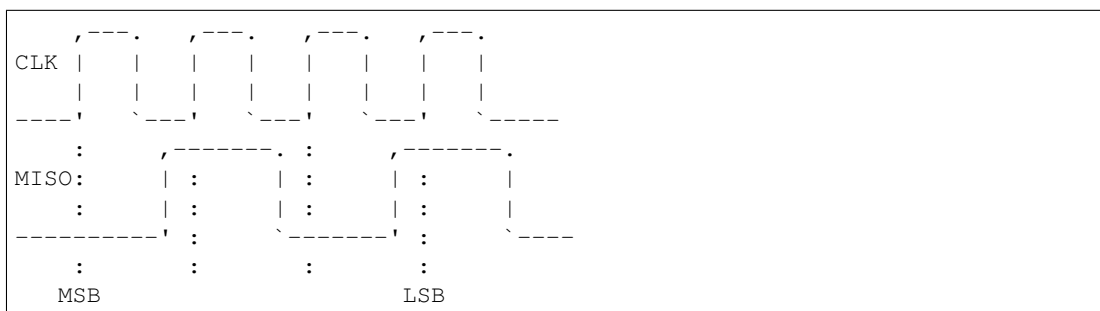
The following diagram illustrates the waveform when `clock_polarity` (page 171) is True, equivalent to CPOL 1:



### `lsb_first`

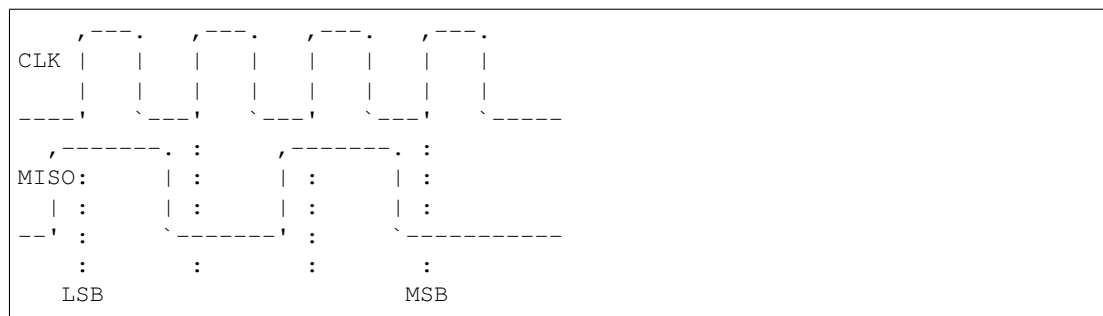
Controls whether words are read and written LSB in (Least Significant Bit first) order. The default is False indicating that words are read and written in MSB (Most Significant Bit first) order. Effectively, this controls the [Bit endianness](https://en.wikipedia.org/wiki/Endianness#Bit_endianness)<sup>413</sup> of the connection.

The following diagram shows the a word containing the number 5 (binary 0101) transmitted on MISO with `bits_per_word` (page 170) set to 4, and `clock_mode` (page 170) set to 0, when `lsb_first` (page 171) is False (the default):



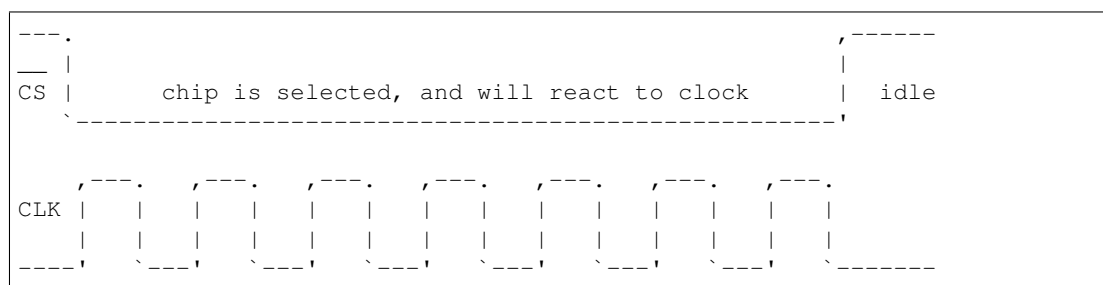
And now with `lsb_first` (page 171) set to True (and all other parameters the same):

<sup>413</sup> [https://en.wikipedia.org/wiki/Endianness#Bit\\_endianness](https://en.wikipedia.org/wiki/Endianness#Bit_endianness)

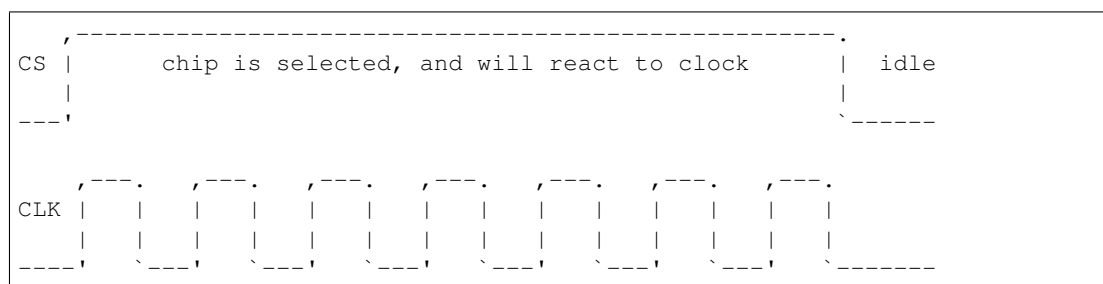
**select\_high**

If `False` (the default), the chip select line is considered active when it is pulled low. When set to `True`, the chip select line is considered active when it is driven high.

The following diagram shows the waveform of the chip select line, and the clock when *clock\_polarity* (page 171) is `False`, and *select\_high* (page 172) is `False` (the default):



And when *select\_high* (page 172) is `True`:

**class** `gpiozero.pins.pi.PiFactory`

Abstract base class representing hardware attached to a Raspberry Pi. This forms the base of *LocalPiFactory* (page 173).

**spi** (*\*\*spi\_args*)

Returns an SPI interface, for the specified SPI *port* and *device*, or for the specified pins (*clock\_pin*, *mosi\_pin*, *miso\_pin*, and *select\_pin*). Only one of the schemes can be used; attempting to mix *port* and *device* with pin numbers will raise `SPIBadArgs`.

If the pins specified match the hardware SPI pins (clock on GPIO11, MOSI on GPIO10, MISO on GPIO9, and chip select on GPIO8 or GPIO7), and the `spidev` module can be imported, a `SPIHardwareInterface` instance will be returned. Otherwise, a `SPISoftwareInterface` will be returned which will use simple bit-banging to communicate.

Both interfaces have the same API, support clock polarity and phase attributes, and can handle half and full duplex communications, but the hardware interface is significantly faster (though for many things this doesn't matter).

**class** `gpiozero.pins.pi.PiPin` (*factory*, *number*)

Abstract base class representing a multi-function GPIO pin attached to a Raspberry Pi. This overrides several methods in the abstract base *Pin* (page 167). Descendents must override the following methods:

- `_get_function()`



- `_set_function()`
- `_get_state()`
- `_call_when_changed()`
- `_enable_event_detect()`
- `_disable_event_detect()`

Descendents *may* additionally override the following methods, if applicable:

- `close()`
- `output_with_state()`
- `input_with_pull()`
- `_set_state()`
- `_get_frequency()`
- `_set_frequency()`
- `_get_pull()`
- `_set_pull()`
- `_get_bounce()`
- `_set_bounce()`
- `_get_edges()`
- `_set_edges()`

**class** `gpiozero.pins.local.LocalPiFactory`

Abstract base class representing pins attached locally to a Pi. This forms the base class for local-only pin interfaces (*RPiGPIOPin* (page 173), *RPIOPin* (page 174), and *NativePin* (page 175)).

**class** `gpiozero.pins.local.LocalPiPin` (*factory*, *number*)

Abstract base class representing a multi-function GPIO pin attached to the local Raspberry Pi.

## RPi.GPIO

**class** `gpiozero.pins.rpigpio.RPiGPIOFactory`

Uses the *RPi.GPIO*<sup>414</sup> 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 RPiGPIOFactory
from gpiozero import LED

factory = RPiGPIOFactory()
led = LED(12, pin_factory=factory)
```

<sup>414</sup> <https://pypi.python.org/pypi/RPi.GPIO>

**class** `gpiozero.pins.rpigpio.RPiGPIOPin` (*factory, number*)  
Pin implementation for the [RPi.GPIO](#)<sup>415</sup> library. See [RPiGPIOFactory](#) (page 173) for more information.

## RPIO

**class** `gpiozero.pins.rpio.RPIOFactory`  
Uses the [RPIO](#)<sup>416</sup> 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 RPIOFactory
from gpiozero import LED

factory = RPIOFactory()
led = LED(12, pin_factory=factory)
```

**class** `gpiozero.pins.rpio.RPiOPin` (*factory, number*)  
Pin implementation for the [RPIO](#)<sup>417</sup> library. See [RPIOFactory](#) (page 174) for more information.

## PiGPIO

**class** `gpiozero.pins.pigpio.PiGPIOFactory` (*host='localhost', port=8888*)  
Uses the [pigpio](#)<sup>418</sup> 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](#)<sup>419</sup> 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 pigpio pins manually like so:

```
from gpiozero.pins.pigpio import PiGPIOFactory
from gpiozero import LED

factory = PiGPIOFactory()
led = LED(12, pin_factory=factory)
```

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:

---

<sup>415</sup> <https://pypi.python.org/pypi/RPi.GPIO>

<sup>416</sup> <https://pythonhosted.org/RPIO/>

<sup>417</sup> <https://pythonhosted.org/RPIO/>

<sup>418</sup> <http://abyz.co.uk/rpi/pigpio/>

<sup>419</sup> <http://abyz.co.uk/rpi/pigpio/>

```
from gpiozero.pins.pigpio import PiGPIOFactory
from gpiozero import LED

factory = PiGPIOFactory(host='192.168.0.2')
led = LED(12, pin_factory=factory)
```

**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.

**class** `gpiozero.pins.pigpio.PiGPIOPin` (*factory, number*)  
Pin implementation for the `pigpio`<sup>420</sup> library. See `PiGPIOFactory` (page 174) for more information.

## Native

**class** `gpiozero.pins.native.NativeFactory`  
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 NativeFactory
from gpiozero import LED

factory = NativeFactory()
led = LED(12, pin_factory=factory)
```

**class** `gpiozero.pins.native.NativePin` (*factory, number*)  
Native pin implementation. See `NativeFactory` (page 175) for more information.

## Mock

**class** `gpiozero.pins.mock.MockFactory` (*revision='a02082', pin\_class=<class 'gpiozero.pins.mock.MockPin'>*)

Factory for generating mock pins. The *revision* parameter specifies what revision of Pi the mock factory pretends to be (this affects the result of the `pi_info` attribute as well as where pull-ups are assumed to be). The *pin\_class* attribute specifies which mock pin class will be generated by the `pin()` (page 175) method by default. This can be changed after construction by modifying the `pin_class` attribute.

**pin** (*spec, pin\_class=None, \*\*kwargs*)

The `pin` method for `MockFactory` (page 175) additionally takes a *pin\_class* attribute which can be used to override the class’ `pin_class` attribute. Any additional keyword arguments will be passed along to the pin constructor (useful with things like `MockConnectedPin` (page 176) which expect to be constructed with another pin).

**reset** ()

Clears the pins and reservations sets. This is primarily useful in test suites to ensure the pin factory is back in a “clean” state before the next set of tests are run.

<sup>420</sup> <http://abyz.co.uk/rpi/pigpio/>

**class** gpiozero.pins.mock.**MockPin** (*factory, number*)

A mock pin used primarily for testing. This class does *not* support PWM.

**class** gpiozero.pins.mock.**MockPWMPin** (*factory, number*)

This derivative of [MockPin](#) (page 175) adds PWM support.

**class** gpiozero.pins.mock.**MockConnectedPin** (*factory, number, input\_pin=None*)

This derivative of [MockPin](#) (page 175) emulates a pin connected to another mock pin. This is used in the “real pins” portion of the test suite to check that one pin can influence another.

**class** gpiozero.pins.mock.**MockChargingPin** (*factory, number, charge\_time=0.01*)

This derivative of [MockPin](#) (page 175) emulates a pin which, when set to input, waits a predetermined length of time and then drives itself high (as if attached to, e.g. a typical circuit using an LDR and a capacitor to time the charging rate).

**class** gpiozero.pins.mock.**MockTriggerPin** (*factory, number, echo\_pin=None, echo\_time=0.04*)

This derivative of [MockPin](#) (page 175) is intended to be used with another [MockPin](#) (page 175) to emulate a distance sensor. Set *echo\_pin* to the corresponding pin instance. When this pin is driven high it will trigger the echo pin to drive high for the echo time.

## CHAPTER 20

---

### API - 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* (page 177), 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* (page 95) you would expect a *ValueError*<sup>421</sup> to be raised. In fact, a *OutputDeviceBadValue* (page 178) error will be raised. However, note that this descends from both *GPIOZeroError* (page 177) (indirectly) and from *ValueError*<sup>422</sup> 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

---

<sup>421</sup> <https://docs.python.org/3.5/library/exceptions.html#ValueError>

<sup>422</sup> <https://docs.python.org/3.5/library/exceptions.html#ValueError>

**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](#) (page 130)

**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](#) (page 103)

**exception** `gpiozero.SPIBadChannel`

Error raised when an invalid channel is given to an [AnalogInputDevice](#) (page 102)

**exception** `gpiozero.SPIFixedClockMode`

Error raised when the SPI clock mode cannot be changed

**exception** `gpiozero.SPIInvalidClockMode`

Error raised when an invalid clock mode is given to an SPI implementation

**exception** `gpiozero.SPIFixedBitOrder`

Error raised when the SPI bit-endianness cannot be changed

**exception** `gpiozero.SPIFixedSelect`

Error raised when the SPI select polarity cannot be changed

**exception** `gpiozero.SPIFixedWordSize`

Error raised when the number of bits per word cannot be changed

**exception** `gpiozero.SPIInvalidWordSize`

Error raised when an invalid (out of range) number of bits per word is specified

**exception** `gpiozero.GPIODeviceError`

Base class for errors specific to the GPIODevice hierarchy

**exception** `gpiozero.GPIODeviceClosed`

Deprecated descendent of [DeviceClosed](#) (page 177)

**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 specification is not given

**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.PinInvalidBounce`

Error raised when attempting to assign an invalid bounce time 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.PinUnsupported`

Error raised when attempting to obtain a pin interface on unsupported pins

**exception** `gpiozero.PinSPIUnsupported`

Error raised when attempting to obtain an SPI interface on unsupported pins

**exception** `gpiozero.PinPWLError`

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.PinUnknownPi`

Error raised when gpiozero doesn't recognize a revision of the Pi

**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.PinInvalidPin`

Error raised when an invalid pin specification is provided

## 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

**exception** `gpiozero.PinFactoryFallback`

Warning raised when a default pin factory fails to load and a fallback is tried

**exception** `gpiozero.PinNonPhysical`

Warning raised when a non-physical pin is specified in a constructor



### Release 1.4.0 (2017-07-26)

- Pin factory is now *configurable from device constructors* (page 164) as well as command line. NOTE: this is a backwards incompatible change for manual pin construction but it's hoped this is (currently) a sufficiently rare use case that this won't affect too many people and the benefits of the new system warrant such a change, i.e. the ability to use remote pin factories with HAT classes that don't accept pin assignments (#279<sup>423</sup>)
- Major work on SPI, primarily to support remote hardware SPI (#421<sup>424</sup>, #459<sup>425</sup>, #465<sup>426</sup>, #468<sup>427</sup>, #575<sup>428</sup>)
- Pin reservation now works properly between GPIO and SPI devices (#459<sup>429</sup>, #468<sup>430</sup>)
- Lots of work on the documentation: *source/values chapter* (page 47), better charts, more recipes, *remote GPIO configuration* (page 35), mock pins, better PDF output (#484<sup>431</sup>, #469<sup>432</sup>, #523<sup>433</sup>, #520<sup>434</sup>, #434<sup>435</sup>, #565<sup>436</sup>, #576<sup>437</sup>)
- Support for *StatusZero* (page 131) and *StatusBoard* (page 133) HATs (#558<sup>438</sup>)
- Added **pinout** command line tool to provide a simple reference to the GPIO layout and information about the associated Pi (#497<sup>439</sup>, #504<sup>440</sup>) thanks to Stewart Adcock for the initial work
- `pi_info()` (page 159) made more lenient for new (unknown) Pi models (#529<sup>441</sup>)

<sup>423</sup> <https://github.com/RPi-Distro/python-gpiozero/issues/279>

<sup>424</sup> <https://github.com/RPi-Distro/python-gpiozero/issues/421>

<sup>425</sup> <https://github.com/RPi-Distro/python-gpiozero/issues/459>

<sup>426</sup> <https://github.com/RPi-Distro/python-gpiozero/issues/465>

<sup>427</sup> <https://github.com/RPi-Distro/python-gpiozero/issues/468>

<sup>428</sup> <https://github.com/RPi-Distro/python-gpiozero/issues/575>

<sup>429</sup> <https://github.com/RPi-Distro/python-gpiozero/issues/459>

<sup>430</sup> <https://github.com/RPi-Distro/python-gpiozero/issues/468>

<sup>431</sup> <https://github.com/RPi-Distro/python-gpiozero/issues/484>

<sup>432</sup> <https://github.com/RPi-Distro/python-gpiozero/issues/469>

<sup>433</sup> <https://github.com/RPi-Distro/python-gpiozero/issues/523>

<sup>434</sup> <https://github.com/RPi-Distro/python-gpiozero/issues/520>

<sup>435</sup> <https://github.com/RPi-Distro/python-gpiozero/issues/434>

<sup>436</sup> <https://github.com/RPi-Distro/python-gpiozero/issues/565>

<sup>437</sup> <https://github.com/RPi-Distro/python-gpiozero/issues/576>

<sup>438</sup> <https://github.com/RPi-Distro/python-gpiozero/issues/558>

<sup>439</sup> <https://github.com/RPi-Distro/python-gpiozero/issues/497>

<sup>440</sup> <https://github.com/RPi-Distro/python-gpiozero/issues/504>

<sup>441</sup> <https://github.com/RPi-Distro/python-gpiozero/issues/529>

- Fixed a variety of packaging issues (#535<sup>442</sup>, #518<sup>443</sup>, #519<sup>444</sup>)
- Improved text in factory fallback warnings (#572<sup>445</sup>)

## Release 1.3.2 (2017-03-03)

- Added new Pi models to stop `pi_info()` (page 159) breaking
- Fix issue with `pi_info()` (page 159) breaking on unknown Pi models

## 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` (page 109) for reading multiple buttons in a single class (#340<sup>446</sup>)
- Added `Servo` (page 88) and `AngularServo` (page 89) classes for controlling simple servo motors (#248<sup>447</sup>)
- Lots of work on supporting easier use of internal and third-party pin implementations (#359<sup>448</sup>)
- `Robot` (page 126) now has a proper `value` (page 127) attribute (#305<sup>449</sup>)
- Added `CPUTemperature` (page 142) as another demo of “internal” devices (#294<sup>450</sup>)
- A temporary work-around for an issue with `DistanceSensor` (page 75) was included but a full fix is in the works (#385<sup>451</sup>)
- More work on the documentation (#320<sup>452</sup>, #295<sup>453</sup>, #289<sup>454</sup>, 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` (page 130) class for controlling Energenie plugs (#69<sup>455</sup>)
- Added `LineSensor` (page 71) class for single line-sensors (#109<sup>456</sup>)

---

<sup>442</sup> <https://github.com/RPi-Distro/python-gpiozero/issues/535>

<sup>443</sup> <https://github.com/RPi-Distro/python-gpiozero/issues/518>

<sup>444</sup> <https://github.com/RPi-Distro/python-gpiozero/issues/519>

<sup>445</sup> <https://github.com/RPi-Distro/python-gpiozero/issues/572>

<sup>446</sup> <https://github.com/RPi-Distro/python-gpiozero/issues/340>

<sup>447</sup> <https://github.com/RPi-Distro/python-gpiozero/issues/248>

<sup>448</sup> <https://github.com/RPi-Distro/python-gpiozero/issues/359>

<sup>449</sup> <https://github.com/RPi-Distro/python-gpiozero/issues/305>

<sup>450</sup> <https://github.com/RPi-Distro/python-gpiozero/issues/294>

<sup>451</sup> <https://github.com/RPi-Distro/python-gpiozero/issues/385>

<sup>452</sup> <https://github.com/RPi-Distro/python-gpiozero/issues/320>

<sup>453</sup> <https://github.com/RPi-Distro/python-gpiozero/issues/295>

<sup>454</sup> <https://github.com/RPi-Distro/python-gpiozero/issues/289>

<sup>455</sup> <https://github.com/RPi-Distro/python-gpiozero/issues/69>

<sup>456</sup> <https://github.com/RPi-Distro/python-gpiozero/issues/109>

- Added *DistanceSensor* (page 75) class for HC-SR04 ultra-sonic sensors (#114<sup>457</sup>)
- Added *SnowPi* (page 134) class for the Ryantek Snow-pi board (#130<sup>458</sup>)
- Added *when\_held* (page 70) (and related properties) to *Button* (page 69) (#115<sup>459</sup>)
- Fixed issues with installing GPIO Zero for python 3 on Raspbian Wheezy releases (#140<sup>460</sup>)
- Added support for lots of ADC chips (MCP3xxx family) (#162<sup>461</sup>) - many thanks to pcpa and lurch!
- Added support for pigpiod as a pin implementation with *PiGPIOPin* (page 175) (#180<sup>462</sup>)
- Many refinements to the base classes mean more consistency in composite devices and several bugs squashed (#164<sup>463</sup>, #175<sup>464</sup>, #182<sup>465</sup>, #189<sup>466</sup>, #193<sup>467</sup>, #229<sup>468</sup>)
- GPIO Zero is now aware of what sort of Pi it's running on via *pi\_info()* (page 159) 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<sup>469</sup>)
- The source/values system was enhanced to ensure normal usage doesn't stress the CPU and lots of utilities were added (#181<sup>470</sup>, #251<sup>471</sup>)

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<sup>472</sup>, #82<sup>473</sup>, #101<sup>474</sup>, #119<sup>475</sup>, #135<sup>476</sup>, #168<sup>477</sup>)
- New *CamJamKitRobot* (page 129) class with the pre-defined motor pins for the new CamJam EduKit
- New *LEDBarGraph* (page 108) class (many thanks to Martin O'Hanlon!) (#126<sup>478</sup>, #176<sup>479</sup>)

<sup>457</sup> <https://github.com/RPi-Distro/python-gpiozero/issues/114>

<sup>458</sup> <https://github.com/RPi-Distro/python-gpiozero/issues/130>

<sup>459</sup> <https://github.com/RPi-Distro/python-gpiozero/issues/115>

<sup>460</sup> <https://github.com/RPi-Distro/python-gpiozero/issues/140>

<sup>461</sup> <https://github.com/RPi-Distro/python-gpiozero/issues/162>

<sup>462</sup> <https://github.com/RPi-Distro/python-gpiozero/issues/180>

<sup>463</sup> <https://github.com/RPi-Distro/python-gpiozero/issues/164>

<sup>464</sup> <https://github.com/RPi-Distro/python-gpiozero/issues/175>

<sup>465</sup> <https://github.com/RPi-Distro/python-gpiozero/issues/182>

<sup>466</sup> <https://github.com/RPi-Distro/python-gpiozero/issues/189>

<sup>467</sup> <https://github.com/RPi-Distro/python-gpiozero/issues/193>

<sup>468</sup> <https://github.com/RPi-Distro/python-gpiozero/issues/229>

<sup>469</sup> <https://github.com/RPi-Distro/python-gpiozero/issues/222>

<sup>470</sup> <https://github.com/RPi-Distro/python-gpiozero/issues/181>

<sup>471</sup> <https://github.com/RPi-Distro/python-gpiozero/issues/251>

<sup>472</sup> <https://github.com/RPi-Distro/python-gpiozero/issues/80>

<sup>473</sup> <https://github.com/RPi-Distro/python-gpiozero/issues/82>

<sup>474</sup> <https://github.com/RPi-Distro/python-gpiozero/issues/101>

<sup>475</sup> <https://github.com/RPi-Distro/python-gpiozero/issues/119>

<sup>476</sup> <https://github.com/RPi-Distro/python-gpiozero/issues/135>

<sup>477</sup> <https://github.com/RPi-Distro/python-gpiozero/issues/168>

<sup>478</sup> <https://github.com/RPi-Distro/python-gpiozero/issues/126>

<sup>479</sup> <https://github.com/RPi-Distro/python-gpiozero/issues/176>

- New `Pin` (page 167) implementation abstracts out the concept of a GPIO pin paving the way for alternate library support and IO extenders in future ([#141](#)<sup>480</sup>)
- New `LEDBoard.blink()` (page 106) method which works properly even when background is set to `False` ([#94](#)<sup>481</sup>, [#161](#)<sup>482</sup>)
- New `RGBLED.blink()` (page 84) method which implements (rudimentary) color fading too! ([#135](#)<sup>483</sup>, [#174](#)<sup>484</sup>)
- New `initial_value` attribute on `OutputDevice` (page 95) ensures consistent behaviour on construction ([#118](#)<sup>485</sup>)
- New `active_high` attribute on `PWMOutputDevice` (page 93) and `RGBLED` (page 84) allows use of common anode devices ([#143](#)<sup>486</sup>, [#154](#)<sup>487</sup>)
- Loads of new ADC chips supported (many thanks to GitHub user pcopa!) ([#150](#)<sup>488</sup>)

## Release 1.0.0 (2015-11-16)

- Debian packaging added ([#44](#)<sup>489</sup>)
- `PWMLLED` (page 82) class added ([#58](#)<sup>490</sup>)
- `TemperatureSensor` removed pending further work ([#93](#)<sup>491</sup>)
- `Buzzer.beep()` (page 86) alias method added ([#75](#)<sup>492</sup>)
- `Motor` (page 87) PWM devices exposed, and `Robot` (page 126) motor devices exposed ([#107](#)<sup>493</sup>)

## Release 0.9.0 (2015-10-25)

Fourth public beta

- Added source and values properties to all relevant classes ([#76](#)<sup>494</sup>)
- Fix names of parameters in `Motor` (page 87) constructor ([#79](#)<sup>495</sup>)
- Added wrappers for LED groups on add-on boards ([#81](#)<sup>496</sup>)

## Release 0.8.0 (2015-10-16)

Third public beta

---

<sup>480</sup> <https://github.com/RPi-Distro/python-gpiozero/issues/141>  
<sup>481</sup> <https://github.com/RPi-Distro/python-gpiozero/issues/94>  
<sup>482</sup> <https://github.com/RPi-Distro/python-gpiozero/issues/161>  
<sup>483</sup> <https://github.com/RPi-Distro/python-gpiozero/issues/135>  
<sup>484</sup> <https://github.com/RPi-Distro/python-gpiozero/issues/174>  
<sup>485</sup> <https://github.com/RPi-Distro/python-gpiozero/issues/118>  
<sup>486</sup> <https://github.com/RPi-Distro/python-gpiozero/issues/143>  
<sup>487</sup> <https://github.com/RPi-Distro/python-gpiozero/issues/154>  
<sup>488</sup> <https://github.com/RPi-Distro/python-gpiozero/issues/150>  
<sup>489</sup> <https://github.com/RPi-Distro/python-gpiozero/issues/44>  
<sup>490</sup> <https://github.com/RPi-Distro/python-gpiozero/issues/58>  
<sup>491</sup> <https://github.com/RPi-Distro/python-gpiozero/issues/93>  
<sup>492</sup> <https://github.com/RPi-Distro/python-gpiozero/issues/75>  
<sup>493</sup> <https://github.com/RPi-Distro/python-gpiozero/issues/107>  
<sup>494</sup> <https://github.com/RPi-Distro/python-gpiozero/issues/76>  
<sup>495</sup> <https://github.com/RPi-Distro/python-gpiozero/issues/79>  
<sup>496</sup> <https://github.com/RPi-Distro/python-gpiozero/issues/81>

- Added generic *AnalogInputDevice* (page 102) class along with specific classes for the *MCP3008* (page 99) and *MCP3004* (page 98) (#41<sup>497</sup>)
- Fixed *DigitalOutputDevice.blink()* (page 92) (#57<sup>498</sup>)

## 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

---

<sup>497</sup> <https://github.com/RPi-Distro/python-gpiozero/issues/41>

<sup>498</sup> <https://github.com/RPi-Distro/python-gpiozero/issues/57>



Copyright 2015-2017 Raspberry Pi Foundation<sup>499</sup>.

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.

---

<sup>499</sup> <https://www.raspberrypi.org/>





### g

- `gpiozero`, 3
- `gpiozero.boards`, 105
- `gpiozero.devices`, 145
- `gpiozero.input_devices`, 69
- `gpiozero.other_devices`, 141
- `gpiozero.output_devices`, 81
- `gpiozero.pins`, 163
  - `gpiozero.pins.local`, 173
  - `gpiozero.pins.mock`, 175
  - `gpiozero.pins.native`, 175
  - `gpiozero.pins.pi`, 172
  - `gpiozero.pins.pigpio`, 174
  - `gpiozero.pins.rpigpio`, 173
  - `gpiozero.pins.rpio`, 174
- `gpiozero.spi_devices`, 97
- `gpiozero.tools`, 151



## Symbols

-c, --color  
pinout command line option, 56

-h, --help  
pinout command line option, 56

-m, --monochrome  
pinout command line option, 56

-r REVISION, --revision REVISION  
pinout command line option, 56

\_shared\_key() (gpiozero.SharedMixin class method), 148

## A

absoluted() (in module gpiozero.tools), 151

active\_high (gpiozero.OutputDevice attribute), 95

active\_time (gpiozero.ButtonBoard attribute), 110

active\_time (gpiozero.EventsMixin attribute), 149

all\_values() (in module gpiozero.tools), 154

alternating\_values() (in module gpiozero.tools), 156

AnalogInputDevice (class in gpiozero), 102

angle (gpiozero.AngularServo attribute), 90

AngularServo (class in gpiozero), 89

any\_values() (in module gpiozero.tools), 155

averaged() (in module gpiozero.tools), 155

## B

backward() (gpiozero.CamJamKitRobot method), 129

backward() (gpiozero.Motor method), 87

backward() (gpiozero.Robot method), 127

backward() (gpiozero.RyanteckRobot method), 128

BadEventHandler, 177

BadQueueLen, 178

BadWaitTime, 178

beep() (gpiozero.Buzzer method), 86

bits (gpiozero.AnalogInputDevice attribute), 103

bits\_per\_word (gpiozero.SPI attribute), 170

blink() (gpiozero.DigitalOutputDevice method), 92

blink() (gpiozero.LED method), 82

blink() (gpiozero.LEDBoard method), 106

blink() (gpiozero.LedBorg method), 114

blink() (gpiozero.PiLiter method), 117

blink() (gpiozero.PiStop method), 122

blink() (gpiozero.PiTraffic method), 120

blink() (gpiozero.PWMLed method), 83

blink() (gpiozero.PWMOutputDevice method), 93

blink() (gpiozero.RGBLED method), 84

blink() (gpiozero.SnowPi method), 135

blink() (gpiozero.StatusZero method), 132

blink() (gpiozero.TrafficLights method), 112

bluetooth (gpiozero.PiBoardInfo attribute), 161

board (gpiozero.PiBoardInfo attribute), 161

booleanized() (in module gpiozero.tools), 151

bounce (gpiozero.Pin attribute), 168

Button (class in gpiozero), 69

ButtonBoard (class in gpiozero), 109

Buzzer (class in gpiozero), 86

## C

CamJamKitRobot (class in gpiozero), 129

channel (gpiozero.MCP3002 attribute), 98

channel (gpiozero.MCP3004 attribute), 98

channel (gpiozero.MCP3008 attribute), 99

channel (gpiozero.MCP3202 attribute), 99

channel (gpiozero.MCP3204 attribute), 99

channel (gpiozero.MCP3208 attribute), 100

channel (gpiozero.MCP3302 attribute), 100

channel (gpiozero.MCP3304 attribute), 101

clamped() (in module gpiozero.tools), 152

clock\_mode (gpiozero.SPI attribute), 170

clock\_phase (gpiozero.SPI attribute), 170

clock\_polarity (gpiozero.SPI attribute), 171

close() (gpiozero.CompositeDevice method), 138

close() (gpiozero.Device method), 147

close() (gpiozero.DigitalOutputDevice method), 92

close() (gpiozero.Energenie method), 130

close() (gpiozero.Factory method), 166

close() (gpiozero.GPIODevice method), 80

close() (gpiozero.LEDBoard method), 106

close() (gpiozero.LedBorg method), 115

close() (gpiozero.PiLiter method), 117

close() (gpiozero.Pin method), 167

close() (gpiozero.PiStop method), 123

close() (gpiozero.PiTraffic method), 121

close() (gpiozero.PWMOutputDevice method), 93

close() (gpiozero.SmoothedInputDevice method), 78

close() (gpiozero.SnowPi method), 135

close() (gpiozero.SPIDevice method), 103

close() (gpiozero.StatusZero method), 132  
 close() (gpiozero.TrafficLights method), 112  
 closed (gpiozero.Device attribute), 147  
 col (gpiozero.PinInfo attribute), 162  
 color (gpiozero.LedBorg attribute), 116  
 color (gpiozero.RGBLED attribute), 85  
 columns (gpiozero.HeaderInfo attribute), 162  
 CompositeDevice (class in gpiozero), 138  
 CompositeDeviceBadName, 178  
 CompositeDeviceError, 178  
 CompositeOutputDevice (class in gpiozero), 137  
 cos\_values() (in module gpiozero.tools), 156  
 CPUTemperature (class in gpiozero), 142  
 csi (gpiozero.PiBoardInfo attribute), 161

## D

detach() (gpiozero.AngularServo method), 90  
 detach() (gpiozero.Servo method), 89  
 Device (class in gpiozero), 147  
 DeviceClosed, 177  
 differential (gpiozero.MCP3002 attribute), 98  
 differential (gpiozero.MCP3004 attribute), 98  
 differential (gpiozero.MCP3008 attribute), 99  
 differential (gpiozero.MCP3202 attribute), 99  
 differential (gpiozero.MCP3204 attribute), 100  
 differential (gpiozero.MCP3208 attribute), 100  
 differential (gpiozero.MCP3302 attribute), 100  
 differential (gpiozero.MCP3304 attribute), 101  
 DigitalInputDevice (class in gpiozero), 77  
 DigitalOutputDevice (class in gpiozero), 92  
 distance (gpiozero.DistanceSensor attribute), 76  
 DistanceSensor (class in gpiozero), 75  
 dsi (gpiozero.PiBoardInfo attribute), 161

## E

echo (gpiozero.DistanceSensor attribute), 76  
 edges (gpiozero.Pin attribute), 168  
 Energenie (class in gpiozero), 130  
 EnergenieBadSocket, 178  
 EnergenieSocketMissing, 178  
 environment variable  
     PIGPIO\_ADDR, 165  
 ethernet (gpiozero.PiBoardInfo attribute), 161  
 EventsMixin (class in gpiozero), 148

## F

Factory (class in gpiozero), 166  
 FishDish (class in gpiozero), 125  
 forward() (gpiozero.CamJamKitRobot method), 129  
 forward() (gpiozero.Motor method), 88  
 forward() (gpiozero.Robot method), 127  
 forward() (gpiozero.RyanteckRobot method), 128  
 frame\_width (gpiozero.AngularServo attribute), 91  
 frame\_width (gpiozero.Servo attribute), 89  
 frequency (gpiozero.Pin attribute), 168  
 frequency (gpiozero.PWMOutputDevice attribute), 94  
 function (gpiozero.Pin attribute), 168  
 function (gpiozero.PinInfo attribute), 162

## G

GPIODevice (class in gpiozero), 80  
 GPIODeviceClosed, 178  
 GPIODeviceError, 178  
 GPIOPinInUse, 178  
 GPIOPinMissing, 178  
 gpiozero (module), 3  
 gpiozero.boards (module), 105  
 gpiozero.devices (module), 145  
 gpiozero.input\_devices (module), 69  
 gpiozero.other\_devices (module), 141  
 gpiozero.output\_devices (module), 81  
 gpiozero.pins (module), 163  
 gpiozero.pins.local (module), 173  
 gpiozero.pins.mock (module), 175  
 gpiozero.pins.native (module), 175  
 gpiozero.pins.pi (module), 172  
 gpiozero.pins.pigpio (module), 174  
 gpiozero.pins.rpigpio (module), 173  
 gpiozero.pins.rpio (module), 174  
 gpiozero.spi\_devices (module), 97  
 gpiozero.tools (module), 151  
 GPIOZeroError, 177  
 GPIOZeroWarning, 179

## H

HeaderInfo (class in gpiozero), 161  
 headers (gpiozero.PiBoardInfo attribute), 161  
 held\_time (gpiozero.Button attribute), 70  
 held\_time (gpiozero.ButtonBoard attribute), 110  
 held\_time (gpiozero.HoldMixin attribute), 149  
 hold\_repeat (gpiozero.Button attribute), 70  
 hold\_repeat (gpiozero.ButtonBoard attribute), 110  
 hold\_repeat (gpiozero.HoldMixin attribute), 149  
 hold\_time (gpiozero.Button attribute), 70  
 hold\_time (gpiozero.ButtonBoard attribute), 110  
 hold\_time (gpiozero.HoldMixin attribute), 149  
 HoldMixin (class in gpiozero), 149

## I

inactive\_time (gpiozero.ButtonBoard attribute), 110  
 inactive\_time (gpiozero.EventsMixin attribute), 149  
 input\_with\_pull() (gpiozero.Pin method), 167  
 InputDevice (class in gpiozero), 79  
 InputDeviceError, 178  
 InternalDevice (class in gpiozero), 143  
 inverted() (in module gpiozero.tools), 152  
 is\_active (gpiozero.Buzzer attribute), 87  
 is\_active (gpiozero.CPUTemperature attribute), 143  
 is\_active (gpiozero.Device attribute), 147  
 is\_active (gpiozero.Energenie attribute), 131  
 is\_active (gpiozero.LedBorg attribute), 116  
 is\_active (gpiozero.PWMOutputDevice attribute), 95  
 is\_active (gpiozero.SmoothedInputDevice attribute), 79  
 is\_held (gpiozero.Button attribute), 70  
 is\_held (gpiozero.ButtonBoard attribute), 110  
 is\_held (gpiozero.HoldMixin attribute), 149  
 is\_lit (gpiozero.LED attribute), 82

is\_lit (gpiozero.LedBorg attribute), 116  
 is\_lit (gpiozero.PWMLED attribute), 83  
 is\_lit (gpiozero.RGBLED attribute), 86  
 is\_pressed (gpiozero.Button attribute), 70

## L

LED (class in gpiozero), 81  
 LEDBarGraph (class in gpiozero), 108  
 LEDBoard (class in gpiozero), 105  
 LedBorg (class in gpiozero), 114  
 LEDCollection (class in gpiozero), 137  
 leds (gpiozero.LEDBarGraph attribute), 108  
 leds (gpiozero.LEDBoard attribute), 107  
 leds (gpiozero.LEDCollection attribute), 137  
 leds (gpiozero.PiLiter attribute), 118  
 leds (gpiozero.PiLiterBarGraph attribute), 119  
 leds (gpiozero.PiStop attribute), 124  
 leds (gpiozero.PiTrafic attribute), 121  
 leds (gpiozero.SnowPi attribute), 136  
 leds (gpiozero.StatusZero attribute), 133  
 leds (gpiozero.TrafficLights attribute), 113  
 left() (gpiozero.CamJamKitRobot method), 129  
 left() (gpiozero.Robot method), 127  
 left() (gpiozero.RyanteckRobot method), 128  
 light\_detected (gpiozero.LightSensor attribute), 75  
 LightSensor (class in gpiozero), 74  
 LineSensor (class in gpiozero), 71  
 LocalPiFactory (class in gpiozero.pins.local), 173  
 LocalPiPin (class in gpiozero.pins.local), 173  
 lsb\_first (gpiozero.SPI attribute), 171

## M

manufacturer (gpiozero.PiBoardInfo attribute), 160  
 max() (gpiozero.AngularServo method), 90  
 max() (gpiozero.Servo method), 89  
 max\_angle (gpiozero.AngularServo attribute), 91  
 max\_distance (gpiozero.DistanceSensor attribute), 76  
 max\_pulse\_width (gpiozero.AngularServo attribute), 91  
 max\_pulse\_width (gpiozero.Servo attribute), 89  
 MCP3001 (class in gpiozero), 98  
 MCP3002 (class in gpiozero), 98  
 MCP3004 (class in gpiozero), 98  
 MCP3008 (class in gpiozero), 99  
 MCP3201 (class in gpiozero), 99  
 MCP3202 (class in gpiozero), 99  
 MCP3204 (class in gpiozero), 99  
 MCP3208 (class in gpiozero), 100  
 MCP3301 (class in gpiozero), 100  
 MCP3302 (class in gpiozero), 100  
 MCP3304 (class in gpiozero), 101  
 memory (gpiozero.PiBoardInfo attribute), 160  
 mid() (gpiozero.AngularServo method), 90  
 mid() (gpiozero.Servo method), 89  
 min() (gpiozero.AngularServo method), 90  
 min() (gpiozero.Servo method), 89  
 min\_angle (gpiozero.AngularServo attribute), 91  
 min\_pulse\_width (gpiozero.AngularServo attribute), 91

min\_pulse\_width (gpiozero.Servo attribute), 89  
 MockChargingPin (class in gpiozero.pins.mock), 176  
 MockConnectedPin (class in gpiozero.pins.mock), 176  
 MockFactory (class in gpiozero.pins.mock), 175  
 MockPin (class in gpiozero.pins.mock), 175  
 MockPWMPin (class in gpiozero.pins.mock), 176  
 MockTriggerPin (class in gpiozero.pins.mock), 176  
 model (gpiozero.PiBoardInfo attribute), 160  
 motion\_detected (gpiozero.MotionSensor attribute), 73  
 MotionSensor (class in gpiozero), 72  
 Motor (class in gpiozero), 87  
 multiplied() (in module gpiozero.tools), 155

## N

name (gpiozero.HeaderInfo attribute), 162  
 NativeFactory (class in gpiozero.pins.native), 175  
 NativePin (class in gpiozero.pins.native), 175  
 negated() (in module gpiozero.tools), 152  
 number (gpiozero.PinInfo attribute), 162

## O

off() (gpiozero.Buzzer method), 86  
 off() (gpiozero.CompositeOutputDevice method), 138  
 off() (gpiozero.DigitalOutputDevice method), 93  
 off() (gpiozero.FishDish method), 125  
 off() (gpiozero.LED method), 82  
 off() (gpiozero.LEDBarGraph method), 108  
 off() (gpiozero.LEDBoard method), 107  
 off() (gpiozero.LedBorg method), 115  
 off() (gpiozero.OutputDevice method), 95  
 off() (gpiozero.PiLiter method), 118  
 off() (gpiozero.PiLiterBarGraph method), 119  
 off() (gpiozero.PiStop method), 123  
 off() (gpiozero.PiTrafic method), 121  
 off() (gpiozero.PWMLED method), 83  
 off() (gpiozero.PWMOutputDevice method), 94  
 off() (gpiozero.RGBLED method), 85  
 off() (gpiozero.SnowPi method), 136  
 off() (gpiozero.StatusBoard method), 134  
 off() (gpiozero.StatusZero method), 133  
 off() (gpiozero.TrafficHat method), 126  
 off() (gpiozero.TrafficLights method), 113  
 off() (gpiozero.TrafficLightsBuzzer method), 124  
 on() (gpiozero.Buzzer method), 87  
 on() (gpiozero.CompositeOutputDevice method), 138  
 on() (gpiozero.DigitalOutputDevice method), 93  
 on() (gpiozero.FishDish method), 125  
 on() (gpiozero.LED method), 82  
 on() (gpiozero.LEDBarGraph method), 108  
 on() (gpiozero.LEDBoard method), 107  
 on() (gpiozero.LedBorg method), 115  
 on() (gpiozero.OutputDevice method), 95  
 on() (gpiozero.PiLiter method), 118  
 on() (gpiozero.PiLiterBarGraph method), 119  
 on() (gpiozero.PiStop method), 123  
 on() (gpiozero.PiTrafic method), 121  
 on() (gpiozero.PWMLED method), 83  
 on() (gpiozero.PWMOutputDevice method), 94

on() (gpiozero.RGBLED method), 85  
on() (gpiozero.SnowPi method), 136  
on() (gpiozero.StatusBoard method), 134  
on() (gpiozero.StatusZero method), 133  
on() (gpiozero.TrafficHat method), 126  
on() (gpiozero.TrafficLights method), 113  
on() (gpiozero.TrafficLightsBuzzer method), 124  
output\_with\_state() (gpiozero.Pin method), 167  
OutputDevice (class in gpiozero), 95  
OutputDeviceBadValue, 178  
OutputDeviceError, 178

## P

partial (gpiozero.SmoothedInputDevice attribute), 79  
pcb\_revision (gpiozero.PiBoardInfo attribute), 160  
physical\_pin() (gpiozero.PiBoardInfo method), 159  
physical\_pins() (gpiozero.PiBoardInfo method), 160  
pi\_info (gpiozero.Factory attribute), 167  
pi\_info() (in module gpiozero), 159  
PiBoardInfo (class in gpiozero), 159  
PiFactory (class in gpiozero.pins.pi), 172  
PIGPIO\_ADDR, 165  
PiGPIOFactory (class in gpiozero.pins.pigpio), 174  
PiGPIOPin (class in gpiozero.pins.pigpio), 175  
PiLiter (class in gpiozero), 116  
PiLiterBarGraph (class in gpiozero), 119  
Pin (class in gpiozero), 167  
pin (gpiozero.Button attribute), 70  
pin (gpiozero.Buzzer attribute), 87  
pin (gpiozero.GPIODevice attribute), 80  
pin (gpiozero.LED attribute), 82  
pin (gpiozero.LightSensor attribute), 75  
pin (gpiozero.LineSensor attribute), 72  
pin (gpiozero.MotionSensor attribute), 73  
pin (gpiozero.PWMLED attribute), 84  
pin() (gpiozero.Factory method), 166  
pin() (gpiozero.pins.mock.MockFactory method), 175  
PinEdgeDetectUnsupported, 179  
PinError, 178  
PinFactoryFallback, 179  
PinFixedPull, 179  
PingServer (class in gpiozero), 142  
PinInfo (class in gpiozero), 162  
PinInvalidBounce, 179  
PinInvalidEdges, 179  
PinInvalidFunction, 179  
PinInvalidPin, 179  
PinInvalidPull, 179  
PinInvalidState, 179  
PinMultiplePins, 179  
PinNonPhysical, 180  
PinNoPins, 179  
pinout command line option  
    -c, --color, 56  
    -h, --help, 56  
    -m, --monochrome, 56  
    -r REVISION, --revision REVISION, 56  
PinPWMError, 179

PinPWMFixedValue, 179  
PinPWMUnsupported, 179  
pins (gpiozero.HeaderInfo attribute), 162  
PinSetInput, 179  
PinSPIUnsupported, 179  
PinUnknownPi, 179  
PinUnsupported, 179  
PiPin (class in gpiozero.pins.pi), 172  
PiStop (class in gpiozero), 122  
PiTraffic (class in gpiozero), 120  
post\_delayed() (in module gpiozero.tools), 152  
post\_periodic\_filtered() (in module gpiozero.tools), 152  
pprint() (gpiozero.HeaderInfo method), 162  
pprint() (gpiozero.PiBoardInfo method), 160  
pre\_delayed() (in module gpiozero.tools), 153  
pre\_periodic\_filtered() (in module gpiozero.tools), 153  
pressed\_time (gpiozero.ButtonBoard attribute), 110  
pull (gpiozero.Pin attribute), 169  
pull\_up (gpiozero.Button attribute), 70  
pull\_up (gpiozero.ButtonBoard attribute), 111  
pull\_up (gpiozero.InputDevice attribute), 80  
pull\_up (gpiozero.PinInfo attribute), 162  
pulled\_up() (gpiozero.PiBoardInfo method), 160  
pulse() (gpiozero.LEDBoard method), 107  
pulse() (gpiozero.LedBorg method), 115  
pulse() (gpiozero.PiLiter method), 118  
pulse() (gpiozero.PiStop method), 123  
pulse() (gpiozero.PiTraffic method), 121  
pulse() (gpiozero.PWMLED method), 83  
pulse() (gpiozero.PWMOutputDevice method), 94  
pulse() (gpiozero.RGBLED method), 85  
pulse() (gpiozero.SnowPi method), 136  
pulse() (gpiozero.StatusZero method), 133  
pulse() (gpiozero.TrafficLights method), 113  
pulse\_width (gpiozero.AngularServo attribute), 91  
pulse\_width (gpiozero.Servo attribute), 89  
PWMLED (class in gpiozero), 82  
PWMOutputDevice (class in gpiozero), 93

## Q

quantized() (in module gpiozero.tools), 153  
queue\_len (gpiozero.SmoothedInputDevice attribute), 79  
queued() (in module gpiozero.tools), 153

## R

random\_values() (in module gpiozero.tools), 156  
raw\_value (gpiozero.AnalogInputDevice attribute), 103  
read() (gpiozero.SPI method), 169  
release\_all() (gpiozero.Factory method), 166  
release\_pins() (gpiozero.Factory method), 166  
released (gpiozero.PiBoardInfo attribute), 160  
reserve\_pins() (gpiozero.Factory method), 166  
reset() (gpiozero.pins.mock.MockFactory method), 175  
reverse() (gpiozero.CamJamKitRobot method), 129  
reverse() (gpiozero.Robot method), 127  
reverse() (gpiozero.RyanteckRobot method), 128  
revision (gpiozero.PiBoardInfo attribute), 160



RGBLED (class in gpiozero), 84  
 right() (gpiozero.CamJamKitRobot method), 129  
 right() (gpiozero.Robot method), 127  
 right() (gpiozero.RyanteckRobot method), 128  
 Robot (class in gpiozero), 126  
 row (gpiozero.PinInfo attribute), 162  
 rows (gpiozero.HeaderInfo attribute), 162  
 RPiGPIOFactory (class in gpiozero.pins.rpigpio), 173  
 RPiGPIOPin (class in gpiozero.pins.rpigpio), 173  
 RPIOFactory (class in gpiozero.pins.rpio), 174  
 RPIOPin (class in gpiozero.pins.rpio), 174  
 RyanteckRobot (class in gpiozero), 128

## S

scaled() (in module gpiozero.tools), 154  
 select\_high (gpiozero.SPI attribute), 172  
 Servo (class in gpiozero), 88  
 SharedMixin (class in gpiozero), 148  
 sin\_values() (in module gpiozero.tools), 156  
 smoothed() (in module gpiozero.tools), 154  
 SmoothedInputDevice (class in gpiozero), 78  
 SnowPi (class in gpiozero), 134  
 soc (gpiozero.PiBoardInfo attribute), 160  
 source (gpiozero.AngularServo attribute), 91  
 source (gpiozero.CamJamKitRobot attribute), 130  
 source (gpiozero.Energenie attribute), 131  
 source (gpiozero.FishDish attribute), 125  
 source (gpiozero.LEDBarGraph attribute), 109  
 source (gpiozero.LEDBoard attribute), 107  
 source (gpiozero.LedBorg attribute), 116  
 source (gpiozero.PiLiter attribute), 118  
 source (gpiozero.PiLiterBarGraph attribute), 119  
 source (gpiozero.PiStop attribute), 124  
 source (gpiozero.PiTraffic attribute), 122  
 source (gpiozero.Robot attribute), 127  
 source (gpiozero.RyanteckRobot attribute), 128  
 source (gpiozero.Servo attribute), 89  
 source (gpiozero.SnowPi attribute), 136  
 source (gpiozero.SourceMixin attribute), 148  
 source (gpiozero.StatusBoard attribute), 134  
 source (gpiozero.StatusZero attribute), 133  
 source (gpiozero.TrafficHat attribute), 126  
 source (gpiozero.TrafficLights attribute), 113  
 source (gpiozero.TrafficLightsBuzzer attribute), 124  
 source\_delay (gpiozero.AngularServo attribute), 91  
 source\_delay (gpiozero.CamJamKitRobot attribute), 130  
 source\_delay (gpiozero.Energenie attribute), 131  
 source\_delay (gpiozero.FishDish attribute), 125  
 source\_delay (gpiozero.LEDBarGraph attribute), 109  
 source\_delay (gpiozero.LEDBoard attribute), 107  
 source\_delay (gpiozero.LedBorg attribute), 116  
 source\_delay (gpiozero.PiLiter attribute), 118  
 source\_delay (gpiozero.PiLiterBarGraph attribute), 119  
 source\_delay (gpiozero.PiStop attribute), 124  
 source\_delay (gpiozero.PiTraffic attribute), 122  
 source\_delay (gpiozero.Robot attribute), 127  
 source\_delay (gpiozero.RyanteckRobot attribute), 129

source\_delay (gpiozero.Servo attribute), 89  
 source\_delay (gpiozero.SnowPi attribute), 136  
 source\_delay (gpiozero.SourceMixin attribute), 148  
 source\_delay (gpiozero.StatusBoard attribute), 134  
 source\_delay (gpiozero.StatusZero attribute), 133  
 source\_delay (gpiozero.TrafficHat attribute), 126  
 source\_delay (gpiozero.TrafficLights attribute), 113  
 source\_delay (gpiozero.TrafficLightsBuzzer attribute), 125  
 SourceMixin (class in gpiozero), 148  
 SPI (class in gpiozero), 169  
 spi() (gpiozero.Factory method), 166  
 spi() (gpiozero.pins.pi.PiFactory method), 172  
 SPIBadArgs, 178  
 SPIBadChannel, 178  
 SPIDevice (class in gpiozero), 103  
 SPIError, 178  
 SPIFixedBitOrder, 178  
 SPIFixedClockMode, 178  
 SPIFixedSelect, 178  
 SPIFixedWordSize, 178  
 SPIInvalidClockMode, 178  
 SPIInvalidWordSize, 178  
 SPISoftwareFallback, 179  
 SPIWarning, 179  
 state (gpiozero.Pin attribute), 169  
 StatusBoard (class in gpiozero), 133  
 StatusZero (class in gpiozero), 131  
 stop() (gpiozero.CamJamKitRobot method), 129  
 stop() (gpiozero.Motor method), 88  
 stop() (gpiozero.Robot method), 127  
 stop() (gpiozero.RyanteckRobot method), 128  
 storage (gpiozero.PiBoardInfo attribute), 161  
 summed() (in module gpiozero.tools), 155

## T

temperature (gpiozero.CPUTemperature attribute), 143  
 threshold (gpiozero.SmoothedInputDevice attribute), 79  
 threshold\_distance (gpiozero.DistanceSensor attribute), 76  
 TimeOfDay (class in gpiozero), 141  
 toggle() (gpiozero.Buzzer method), 87  
 toggle() (gpiozero.CompositeOutputDevice method), 138  
 toggle() (gpiozero.FishDish method), 125  
 toggle() (gpiozero.LED method), 82  
 toggle() (gpiozero.LEDBarGraph method), 108  
 toggle() (gpiozero.LEDBoard method), 107  
 toggle() (gpiozero.LedBorg method), 116  
 toggle() (gpiozero.OutputDevice method), 95  
 toggle() (gpiozero.PiLiter method), 118  
 toggle() (gpiozero.PiLiterBarGraph method), 119  
 toggle() (gpiozero.PiStop method), 124  
 toggle() (gpiozero.PiTraffic method), 121  
 toggle() (gpiozero.PWMLED method), 83  
 toggle() (gpiozero.PWMOutputDevice method), 94  
 toggle() (gpiozero.RGBLED method), 85

toggle() (gpiozero.SnowPi method), 136  
 toggle() (gpiozero.StatusBoard method), 134  
 toggle() (gpiozero.StatusZero method), 133  
 toggle() (gpiozero.TrafficHat method), 126  
 toggle() (gpiozero.TrafficLights method), 113  
 toggle() (gpiozero.TrafficLightsBuzzer method), 124  
 TrafficHat (class in gpiozero), 126  
 TrafficLights (class in gpiozero), 111  
 TrafficLightsBuzzer (class in gpiozero), 124  
 transfer() (gpiozero.SPI method), 170  
 trigger (gpiozero.DistanceSensor attribute), 77

## U

usb (gpiozero.PiBoardInfo attribute), 161

## V

value (gpiozero.AnalogInputDevice attribute), 103  
 value (gpiozero.AngularServo attribute), 91  
 value (gpiozero.CamJamKitRobot attribute), 130  
 value (gpiozero.CompositeOutputDevice attribute), 138  
 value (gpiozero.Device attribute), 147  
 value (gpiozero.FishDish attribute), 125  
 value (gpiozero.LEDBarGraph attribute), 109  
 value (gpiozero.LEDBoard attribute), 107  
 value (gpiozero.LedBorg attribute), 116  
 value (gpiozero.MCP3001 attribute), 98  
 value (gpiozero.MCP3002 attribute), 98  
 value (gpiozero.MCP3004 attribute), 99  
 value (gpiozero.MCP3008 attribute), 99  
 value (gpiozero.MCP3201 attribute), 99  
 value (gpiozero.MCP3202 attribute), 99  
 value (gpiozero.MCP3204 attribute), 100  
 value (gpiozero.MCP3208 attribute), 100  
 value (gpiozero.MCP3301 attribute), 100  
 value (gpiozero.MCP3302 attribute), 101  
 value (gpiozero.MCP3304 attribute), 101  
 value (gpiozero.OutputDevice attribute), 95  
 value (gpiozero.PiLiter attribute), 118  
 value (gpiozero.PiLiterBarGraph attribute), 119  
 value (gpiozero.PiStop attribute), 124  
 value (gpiozero.PiTraffic attribute), 122  
 value (gpiozero.PWMLed attribute), 84  
 value (gpiozero.PWMOutputDevice attribute), 95  
 value (gpiozero.Robot attribute), 127  
 value (gpiozero.RyanteckRobot attribute), 129  
 value (gpiozero.Servo attribute), 89  
 value (gpiozero.SmoothedInputDevice attribute), 79  
 value (gpiozero.SnowPi attribute), 136  
 value (gpiozero.StatusBoard attribute), 134  
 value (gpiozero.StatusZero attribute), 133  
 value (gpiozero.TrafficHat attribute), 126  
 value (gpiozero.TrafficLights attribute), 114  
 value (gpiozero.TrafficLightsBuzzer attribute), 125  
 values (gpiozero.AngularServo attribute), 91  
 values (gpiozero.ButtonBoard attribute), 111  
 values (gpiozero.CamJamKitRobot attribute), 130  
 values (gpiozero.Energenie attribute), 131  
 values (gpiozero.FishDish attribute), 125

values (gpiozero.LEDBarGraph attribute), 109  
 values (gpiozero.LEDBoard attribute), 107  
 values (gpiozero.LedBorg attribute), 116  
 values (gpiozero.PiLiter attribute), 118  
 values (gpiozero.PiLiterBarGraph attribute), 120  
 values (gpiozero.PiStop attribute), 124  
 values (gpiozero.PiTraffic attribute), 122  
 values (gpiozero.Robot attribute), 128  
 values (gpiozero.RyanteckRobot attribute), 129  
 values (gpiozero.Servo attribute), 89  
 values (gpiozero.SnowPi attribute), 136  
 values (gpiozero.StatusBoard attribute), 134  
 values (gpiozero.StatusZero attribute), 133  
 values (gpiozero.TrafficHat attribute), 126  
 values (gpiozero.TrafficLights attribute), 114  
 values (gpiozero.TrafficLightsBuzzer attribute), 125  
 values (gpiozero.ValuesMixin attribute), 148  
 ValuesMixin (class in gpiozero), 147

## W

wait\_for\_active() (gpiozero.ButtonBoard method), 110  
 wait\_for\_active() (gpiozero.EventsMixin method), 148  
 wait\_for\_dark() (gpiozero.LightSensor method), 74  
 wait\_for\_in\_range() (gpiozero.DistanceSensor method), 76  
 wait\_for\_inactive() (gpiozero.ButtonBoard method), 110  
 wait\_for\_inactive() (gpiozero.EventsMixin method), 149  
 wait\_for\_light() (gpiozero.LightSensor method), 75  
 wait\_for\_line() (gpiozero.LineSensor method), 72  
 wait\_for\_motion() (gpiozero.MotionSensor method), 73  
 wait\_for\_no\_line() (gpiozero.LineSensor method), 72  
 wait\_for\_no\_motion() (gpiozero.MotionSensor method), 73  
 wait\_for\_out\_of\_range() (gpiozero.DistanceSensor method), 76  
 wait\_for\_press() (gpiozero.Button method), 70  
 wait\_for\_press() (gpiozero.ButtonBoard method), 110  
 wait\_for\_release() (gpiozero.Button method), 70  
 wait\_for\_release() (gpiozero.ButtonBoard method), 110  
 when\_activated (gpiozero.ButtonBoard attribute), 111  
 when\_activated (gpiozero.EventsMixin attribute), 149  
 when\_changed (gpiozero.Pin attribute), 169  
 when\_dark (gpiozero.LightSensor attribute), 75  
 when\_deactivated (gpiozero.ButtonBoard attribute), 111  
 when\_deactivated (gpiozero.EventsMixin attribute), 149  
 when\_held (gpiozero.Button attribute), 70  
 when\_held (gpiozero.ButtonBoard attribute), 111  
 when\_held (gpiozero.HoldMixin attribute), 149  
 when\_in\_range (gpiozero.DistanceSensor attribute), 77  
 when\_light (gpiozero.LightSensor attribute), 75  
 when\_line (gpiozero.LineSensor attribute), 72  
 when\_motion (gpiozero.MotionSensor attribute), 73



`when_no_line` (gpiozero.LineSensor attribute), [72](#)  
`when_no_motion` (gpiozero.MotionSensor attribute),  
[73](#)  
`when_out_of_range` (gpiozero.DistanceSensor attribute), [77](#)  
`when_pressed` (gpiozero.Button attribute), [71](#)  
`when_pressed` (gpiozero.ButtonBoard attribute), [111](#)  
`when_released` (gpiozero.Button attribute), [71](#)  
`when_released` (gpiozero.ButtonBoard attribute), [111](#)  
`wifi` (gpiozero.PiBoardInfo attribute), [161](#)  
`write()` (gpiozero.SPI method), [170](#)