
Gpiozero Documentation

Release 1.4.1

Ben Nuttall

Jun 13, 2018

Contents

1	Installing GPIO Zero	1
2	Basic Recipes	3
3	Advanced Recipes	27
4	Configuring Remote GPIO	35
5	Remote GPIO Recipes	43
6	Pi Zero USB OTG	47
7	Source/Values	51
8	Command-line Tools	57
9	Frequently Asked Questions	65
10	Contributing	69
11	Development	71
12	API - Input Devices	73
13	API - Output Devices	87
14	API - SPI Devices	105
15	API - Boards and Accessories	113
16	API - Internal Devices	155
17	API - Generic Classes	159
18	API - Device Source Tools	165
19	API - Pi Information	173
20	API - Pins	177
21	API - Exceptions	191
22	Changelog	195

23 License	201
Python Module Index	203

CHAPTER 1

Installing GPIO Zero

GPIO Zero is installed by default in the [Raspbian](https://www.raspberrypi.org/downloads/raspbian/)¹ image, and the [Raspberry Pi Desktop](https://www.raspberrypi.org/downloads/raspberry-pi-desktop/)² image for PC/Mac, both available from [raspberrypi.org](https://www.raspberrypi.org)³. Follow these guides to installing on Raspbian Lite and other operating systems, including for PCs using the *remote GPIO* (page 35) feature.

1.1 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](https://pip.pypa.io/en/stable/installing/)⁴ 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 71) page.

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

² <https://www.raspberrypi.org/downloads/raspberry-pi-desktop/>

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

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

1.2 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!

2.1 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 73) interface from GPIO Zero, it should be explicitly imported:

```
from gpiozero import Button
```

Now *Button* (page 73) 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)
```

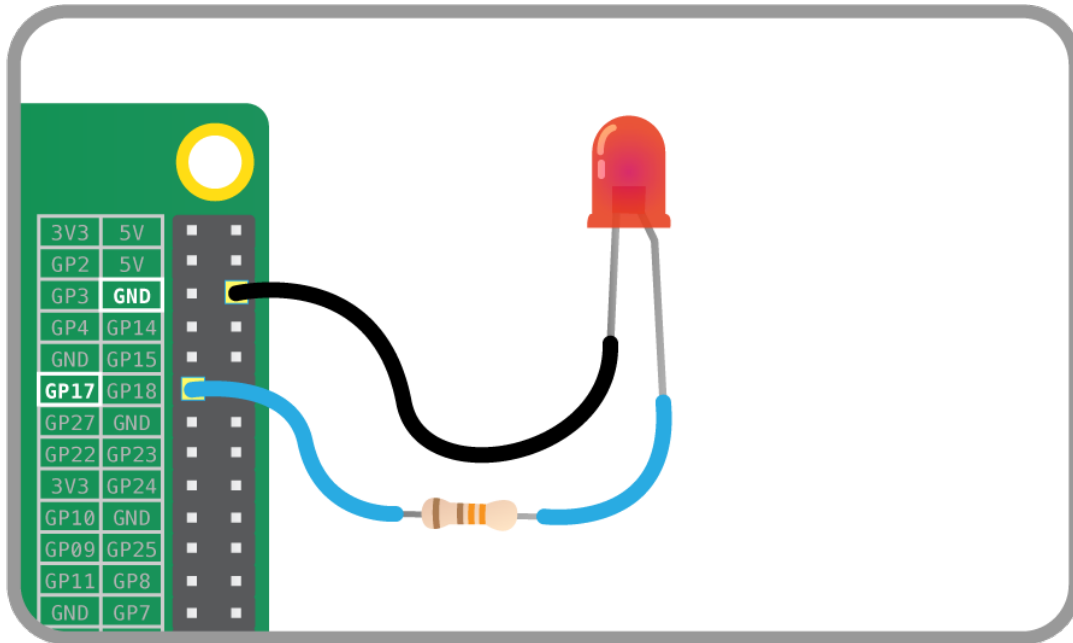
2.2 Pin Numbering

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

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

2.3 LED



Turn an [LED](#) (page 87) 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()`⁶. See [How do I keep my script running?](#) (page 65) for more information.

2.4 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 88) using values between 0 and 1:

⁶ <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

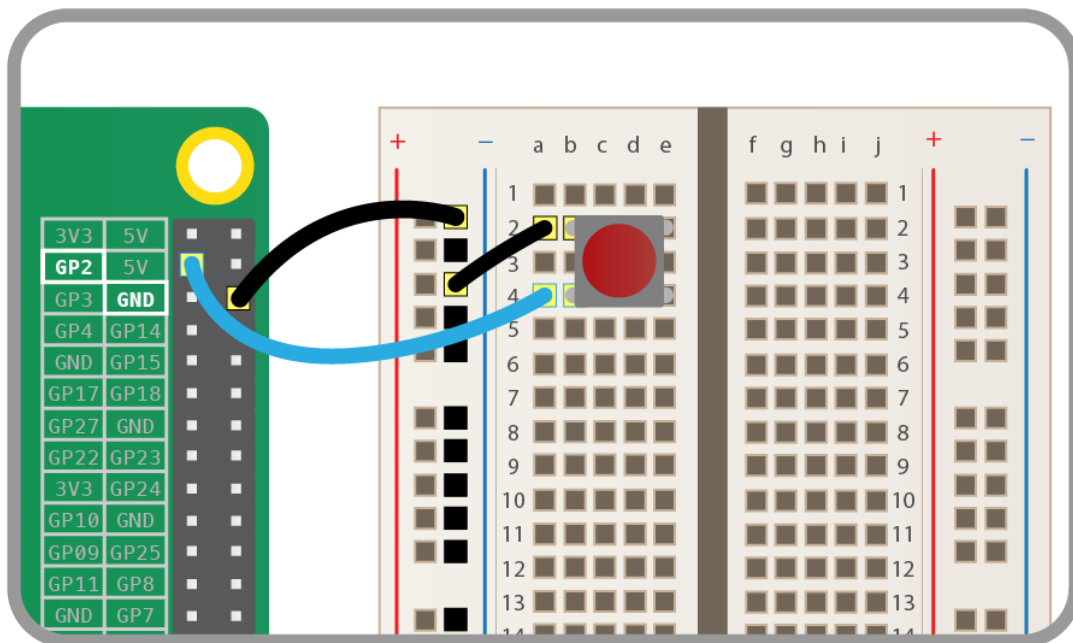
led = PWMLED(17)

led.pulse()

pause()

```

2.5 Button



Check if a *Button* (page 73) 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!")

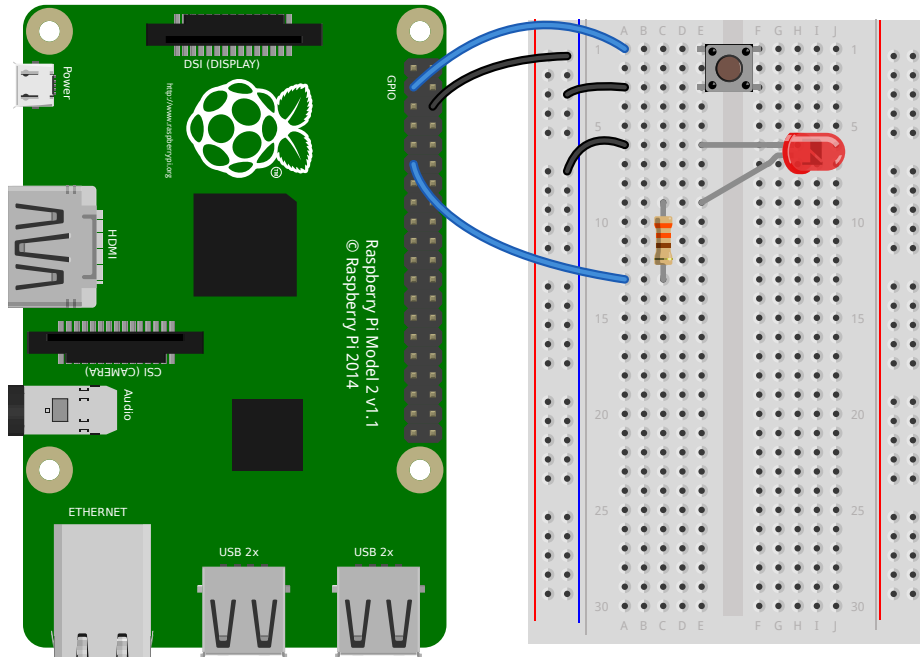
def say_goodbye():
    print("Goodbye!")

button = Button(2)

button.when_pressed = say_hello
button.when_released = say_goodbye

pause()
```

2.6 Button controlled LED



Turn on an [LED](#) (page 87) when a [Button](#) (page 73) 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()
```

2.7 Button controlled camera

Using the button press to trigger [PiCamera](#)⁷ to take a picture using `button.when_pressed = camera.capture` would not work because the `capture()`⁸ method requires an output parameter. However, this can be achieved using a custom function which requires no parameters:

⁷ https://picamera.readthedocs.io/en/latest/api_camera.html#picamera.PiCamera

⁸ 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()

```

2.8 Shutdown button

The *Button* (page 73) 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()

```

2.9 LEDBoard

A collection of LEDs can be accessed using *LEDBoard* (page 113):

```
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 113) 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 113) examples in the *advanced LEDBoard recipes* (page 27).

2.10 LEDBarGraph

A collection of LEDs can be treated like a bar graph using *LEDBarGraph* (page 116):

```
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 116) 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)
```

(continues on next page)

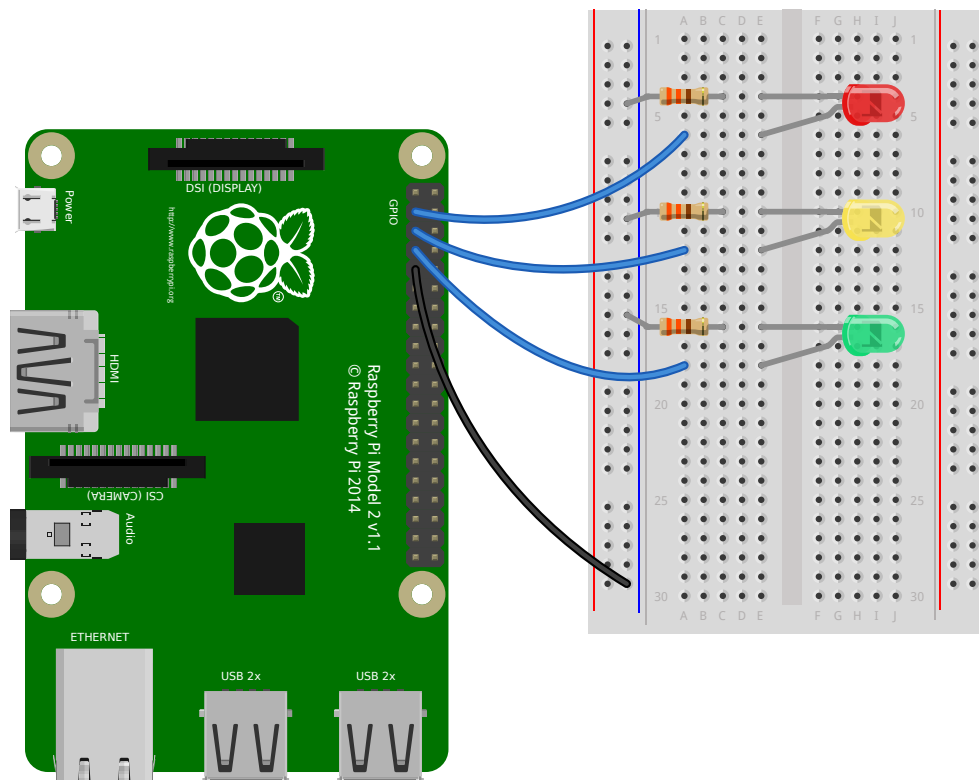
(continued from previous page)

```

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)

```

2.11 Traffic Lights



A full traffic lights system.

Using a *TrafficLights* (page 120) 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)

```

(continues on next page)

(continued from previous page)

```
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 87) 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()
```

2.12 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

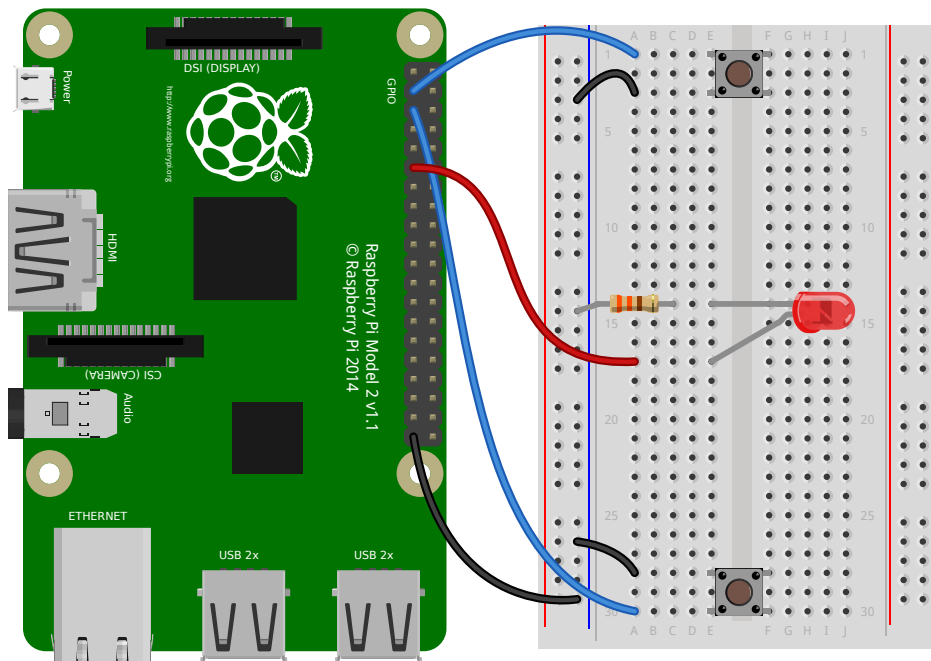
button = Button(2)
camera = PiCamera()

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 Motion⁹](#) for a full resource.

2.13 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!")

```

(continues on next page)

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

(continued from previous page)

```
        break
    if player_2.is_pressed:
        print("Player 2 wins!")
        break

led.off()
```

See [Quick Reaction Game](#)¹⁰ for a full resource.

2.14 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](#)¹¹ for a full resource.

2.15 All on when pressed

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

FishDish (page 134):

```
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 135):

```
from gpiozero import TrafficHat
from signal import pause

th = TrafficHat()
```

(continues on next page)

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

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

(continued from previous page)

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

pause()
```

Using [LED](#) (page 87), [Buzzer](#) (page 92), and [Button](#) (page 73) 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]

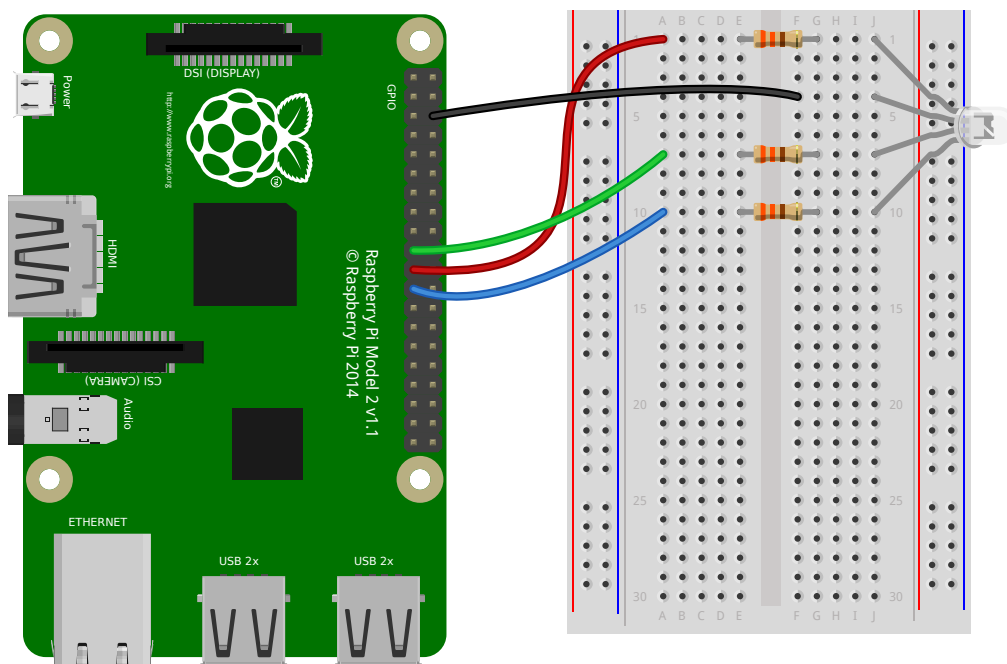
def things_on():
    for thing in things:
        thing.on()

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

button.when_pressed = things_on
button.when_released = things_off

pause()
```

2.16 Full color LED



Making colours with an [RGBLED](#) (page 90):

```
from gpiozero import RGBLED
from time import sleep

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

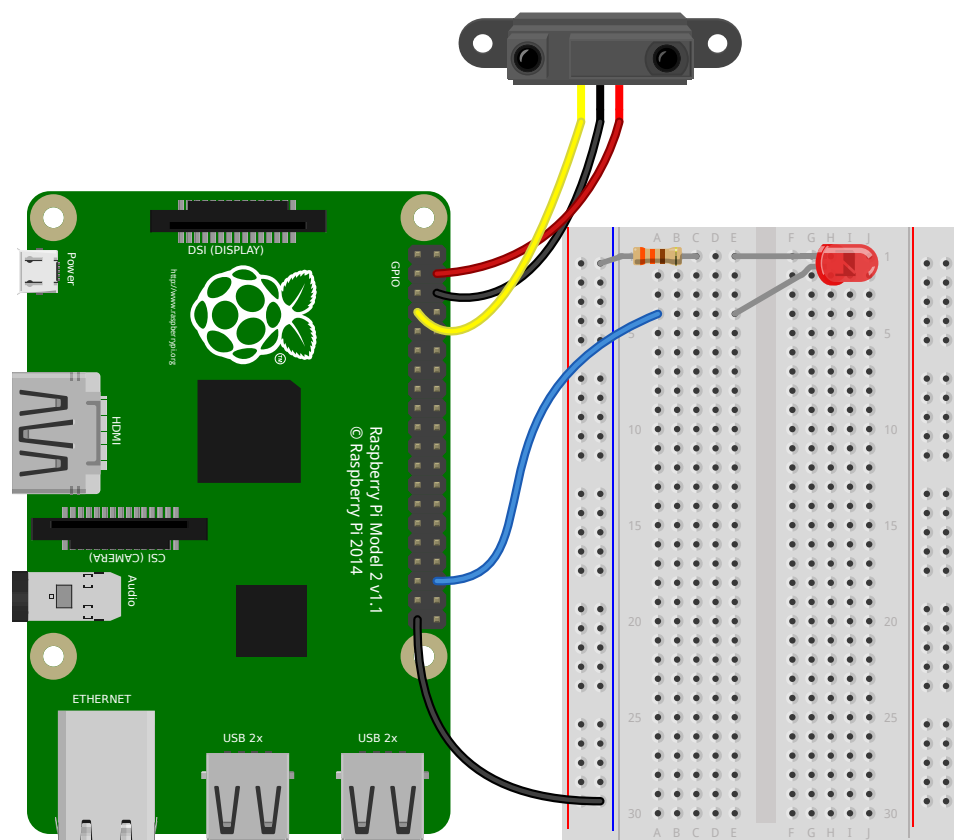
led.red = 1 # full red
sleep(1)
led.red = 0.5 # half red
sleep(1)

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)
```

2.17 Motion sensor



Light an *LED* (page 87) when a *MotionSensor* (page 76) detects motion:

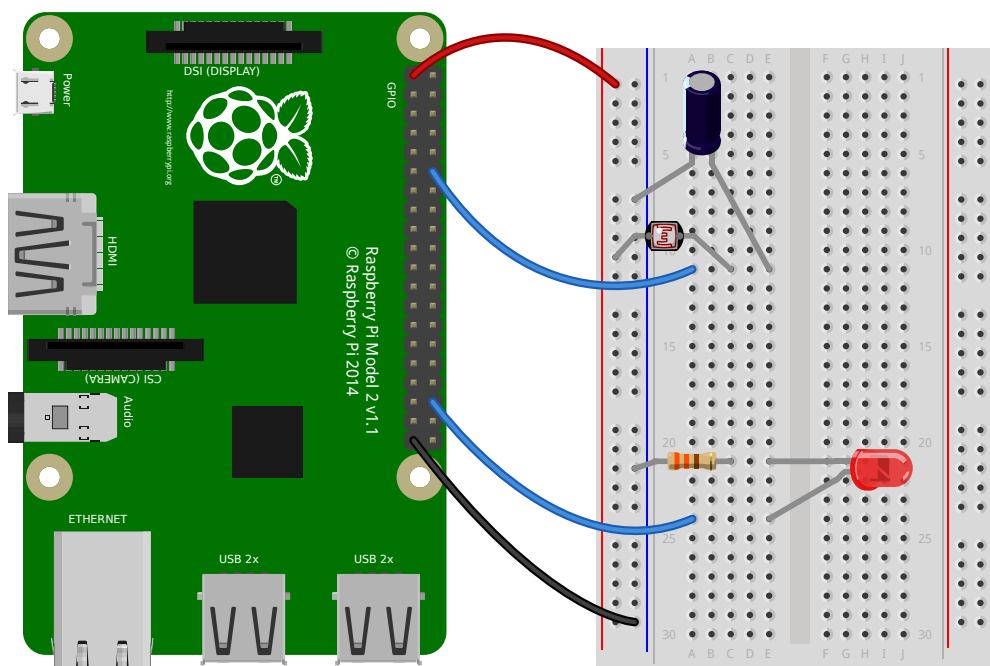
```
from gpiozero import MotionSensor, LED
from signal import pause

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

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

pause()
```

2.18 Light sensor



Have a *LightSensor* (page 78) 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
```

(continues on next page)

(continued from previous page)

```
sensor.when_light = led.off
pause()
```

Or make a [PWMLED](#) (page 88) change brightness according to the detected light level:

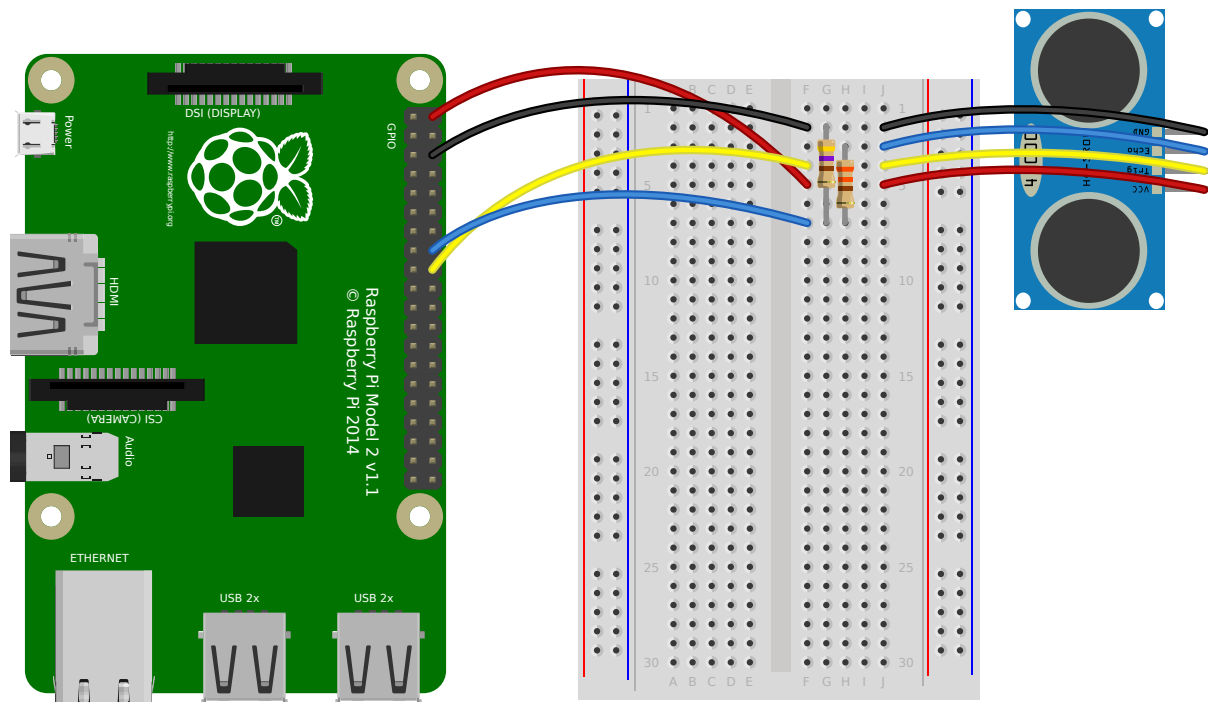
```
from gpiozero import LightSensor, PWMLED
from signal import pause

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

led.source = sensor.values

pause()
```

2.19 Distance sensor



Note: In the diagram above, the wires leading from the sensor to the breadboard can be omitted; simply plug the sensor directly into the breadboard facing the edge (unfortunately this is difficult to illustrate in the diagram without sensor's diagram obscuring most of the breadboard!)

Have a [DistanceSensor](#) (page 79) 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:

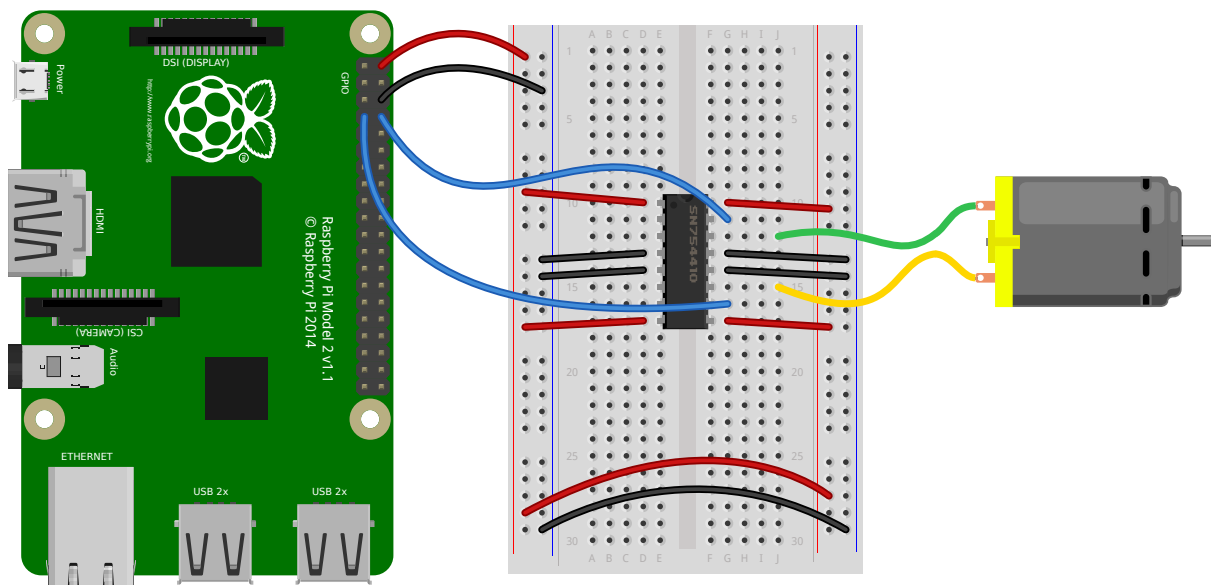
```
from gpiozero import DistanceSensor, LED
from signal import pause

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()
```

2.20 Motors



Spin a *Motor* (page 93) 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)
```

2.21 Robot

Make a *Robot* (page 136) drive around in (roughly) a square:

```
from gpiozero import Robot
from time import sleep

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

(continues on next page)

(continued from previous page)

```
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()
```

2.22 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()
```

2.23 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 = {
```

(continues on next page)

(continued from previous page)

```

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`¹² 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 = {

```

(continues on next page)

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

(continued from previous page)

```
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.

2.24 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:

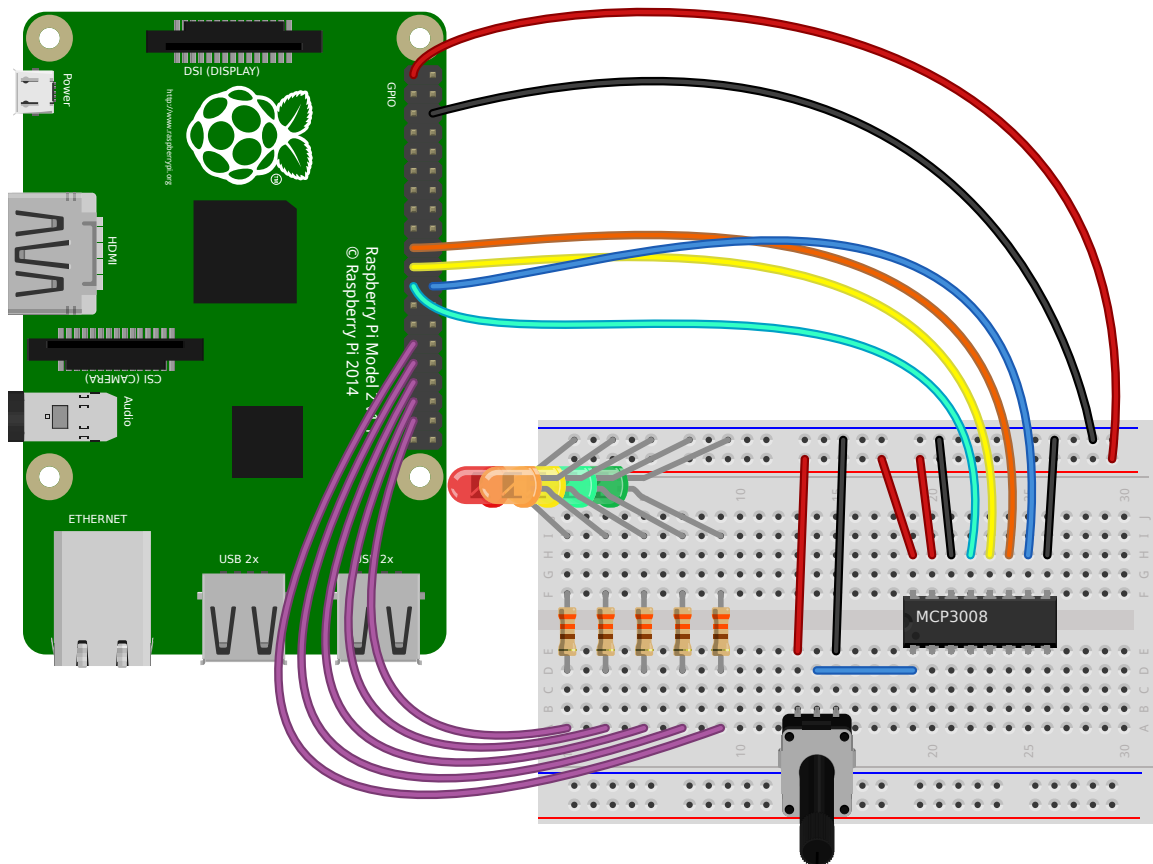
```
from gpiozero import Robot, MotionSensor
from signal import pause

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

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

pause()
```

2.25 Potentiometer



Continually print the value of a potentiometer (values between 0 and 1) connected to a *MCP3008* (page 107) 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()
```

2.26 Measure temperature with an ADC

Wire a TMP36 temperature sensor to the first channel of an *MCP3008* (page 107) analog to digital converter:

```
from gpiozero import MCP3008
from time import sleep
```

(continues on next page)

(continued from previous page)

```
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)
```

2.27 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 162) property rather than a `while`¹³ loop:

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

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

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

pause()
```

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

2.28 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](#)¹⁵ to remotely control the lamp, and the [TimeOfDay](#) (page 155) class to control the timing:

¹³ https://docs.python.org/3.5/reference/compound_stmts.html#while

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

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

```

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()

```

2.29 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 156) 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(google.values)

pause()

```

2.30 CPU Temperature Bar Graph

You can read the Raspberry Pi’s own CPU temperature using the built-in *CPUTemperature* (page 156) 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()

```

2.31 More recipes

Continue to:

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

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!

3.1 LEDBoard

You can iterate over the LEDs in a *LEDBoard* (page 113) 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 113) also supports indexing. This means you can access the individual *LED* (page 87) 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
```

(continues on next page)

(continued from previous page)

```
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 113) 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 113) objects can also be nested within other `LEDBoard` (page 113) 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)
```


3.2 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 Zero¹⁶](#) 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()
```

3.3 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
```

(continues on next page)

¹⁶ <https://thepihut.com/status>

(continued from previous page)

```
from gpiozero.tools import negated
from time import sleep
from signal import pause

def build_passed(repo):
    t = TravisPy()
    r = t.repo(repo)
    while True:
        yield r.last_build_state == 'passed'

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

green.source = build_passed('RPi-Distro/python-gpiozero')
green.source_delay = 60 * 5 # check every 5 minutes
red.source = negated(green.values)

pause()
```

Note this recipe requires `travispy`¹⁷. Install with `sudo pip3 install travispy`.

3.4 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):
```

(continues on next page)

¹⁷ <https://travispy.readthedocs.io/>

(continued from previous page)

```
button.when_pressed = add_instruction
pause()
```

3.5 Robot controlled by 2 potentiometers

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

```
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()`¹⁸ 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()
```

3.6 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 bluepy import BlueDot
from gpiozero import LED

bd = BlueDot()
led = LED(17)

while True:
    bd.wait_for_press()
    led.on()
```

(continues on next page)

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

(continued from previous page)

```
bd.wait_for_release()
led.off()
```

Note this recipe requires `bluedot` and the associated Android app. See the [BlueDot documentation](#)¹⁹ for installation instructions.

3.7 BlueDot robot

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

```
from bluedot 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 bluedot 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))
```

(continues on next page)

¹⁹ <https://bluedot.readthedocs.io/en/latest/index.html>

(continued from previous page)

```
bd = BlueDot()

robot.source = drive()

pause()
```

3.8 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](#)²⁰ 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 177) documentation page.

One of the pin libraries supported, [pigpio](#)²¹, 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.

4.1 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.me.uk](#)²².

You'll need to enable remote connections, and launch the pigpio daemon on the Raspberry Pi.

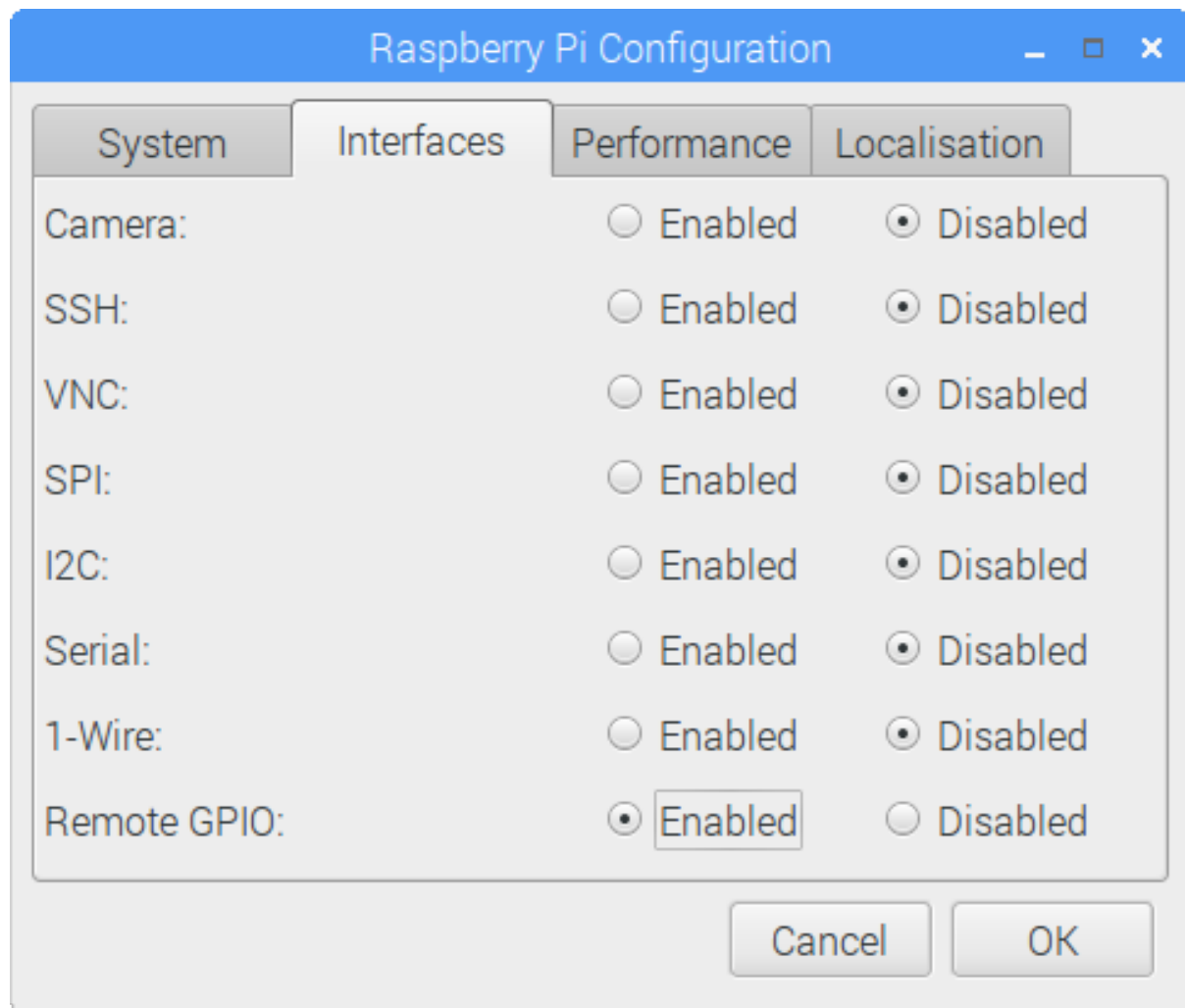
4.1.1 Enable remote connections

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

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

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

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



Alternatively, enter `sudo raspi-config` on the command line, and enable Remote GPIO. This is functionally equivalent to the desktop method.

This will allow remote connections (until disabled) when the `pigpio` daemon is launched using `systemctl` (see below). It will also launch the `pigpio` daemon for the current session. Therefore, nothing further is required for the current session, but after a reboot, a `systemctl` command will be required.

4.1.2 Command-line: `systemctl`

To automate running the daemon at boot time, run:

```
$ sudo systemctl enable pigpiod
```

To run the daemon once using `systemctl`, run:

```
$ sudo systemctl start pigpiod
```

4.1.3 Command-line: `pigpiod`

Another option is to launch the `pigpio` daemon manually:

```
$ sudo pigpiod
```

This is for single-session-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
```

Note: Note that running `sudo pigpiod` will not honour the Remote GPIO configuration setting (i.e. without the `-n` flag it will allow remote connections even if the remote setting is disabled), but `sudo systemctl enable pigpiod` or `sudo systemctl start pigpiod` will not allow remote connections unless configured accordingly.

4.2 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 [Raspberry Pi Desktop x86²³](https://www.raspberrypi.org/downloads/raspberry-pi-desktop/)), 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.

4.2.1 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
```

4.2.2 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
```

²³ <https://www.raspberrypi.org/downloads/raspberry-pi-desktop/>

(Alternatively, install pip with [get-pip](#)²⁴.)

Next, install GPIO Zero and pigpio for Python 3:

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

or Python 2:

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

4.2.3 Mac OS

First, install pip. If you installed Python 3 using brew, you will already have pip. If not, install pip with [get-pip](#)²⁵.

Next, install GPIO Zero and pigpio with pip:

```
$ pip3 install gpiozero pigpio
```

Or for Python 2:

```
$ pip install gpiozero pigpio
```

4.2.4 Windows

First, install pip by [following this guide](#)²⁶. Next, install GPIO Zero and pigpio with pip:

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

4.3 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:

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

²⁵ <https://pip.pypa.io/en/stable/installing/>

²⁶ <https://www.raspberrypi.org/learning/using-pip-on-windows/worksheet/>

```

from gpiozero import LED
from time import sleep

red = LED(17)

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

```

When run with:

```
$ 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.

4.4 Pin factories

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:

```

(continues on next page)

(continued from previous page)

```
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 `GPIZERO_PIN_FACTORY=pigpio` `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.rpigpio 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:
```

(continues on next page)

(continued from previous page)

```

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 87) 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`²⁷) 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.

4.5 Remote GPIO usage

Continue to:

- *Remote GPIO Recipes* (page 43)

²⁷ <https://www.raspberrypi.org/products/sense-hat/>

- *Pi Zero USB OTG* (page 47)

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!

5.1 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()
```

5.2 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')
```

(continues on next page)

(continued from previous page)

```
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()
```

5.3 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()
```

5.4 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).

5.5 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()

```

5.6 Light sensor + Sense HAT

The [Sense HAT](https://www.raspberrypi.org/products/sense-hat/)²⁸ (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.

²⁸ <https://www.raspberrypi.org/products/sense-hat/>

Pi Zero USB OTG

The [Raspberry Pi Zero](#)²⁹ and [Pi Zero W](#)³⁰ 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 the *remote GPIO* (page 35) feature.

6.1 GPIO expander method - no SD card required

The GPIO expander method allows you to boot the Pi Zero over USB from the PC, without an SD card. Your PC sends the required boot firmware to the Pi over the USB cable, launching a mini version of Raspbian and booting it in RAM. The OS then starts the *pigpio* daemon, allowing “remote” access over the USB cable.

At the time of writing, this is only possible using either the Raspberry Pi Desktop x86 OS, or Ubuntu (or a derivative), or from another Raspberry Pi. Usage from Windows and Mac OS is not supported at present.

6.1.1 Raspberry Pi Desktop x86 setup

1. Download an ISO of the [Raspberry Pi Desktop OS](#)³¹ from [raspberrypi.org](#) (this must be the Stretch release, not the older Jessie image).
2. Write the image to a USB stick or burn to a DVD.
3. Live boot your PC or Mac into the OS (select “Run with persistence” and your computer will be back to normal afterwards).

6.1.2 Raspberry Pi (Raspbian) setup

1. Update your package list and install the `usbbootgui` package:

```
$ sudo apt update
$ sudo apt install usbbootgui
```

²⁹ <https://www.raspberrypi.org/products/raspberry-pi-zero/>

³⁰ <https://www.raspberrypi.org/products/raspberry-pi-zero-w/>

³¹ <https://www.raspberrypi.org/downloads/raspberry-pi-desktop/>

6.1.3 Ubuntu setup

1. Add the Raspberry Pi PPA to your system:

```
$ sudo add-apt-repository ppa:rpi-distro/ppa
```

2. If you have previously installed gpiozero or pigpio with pip, uninstall these first:

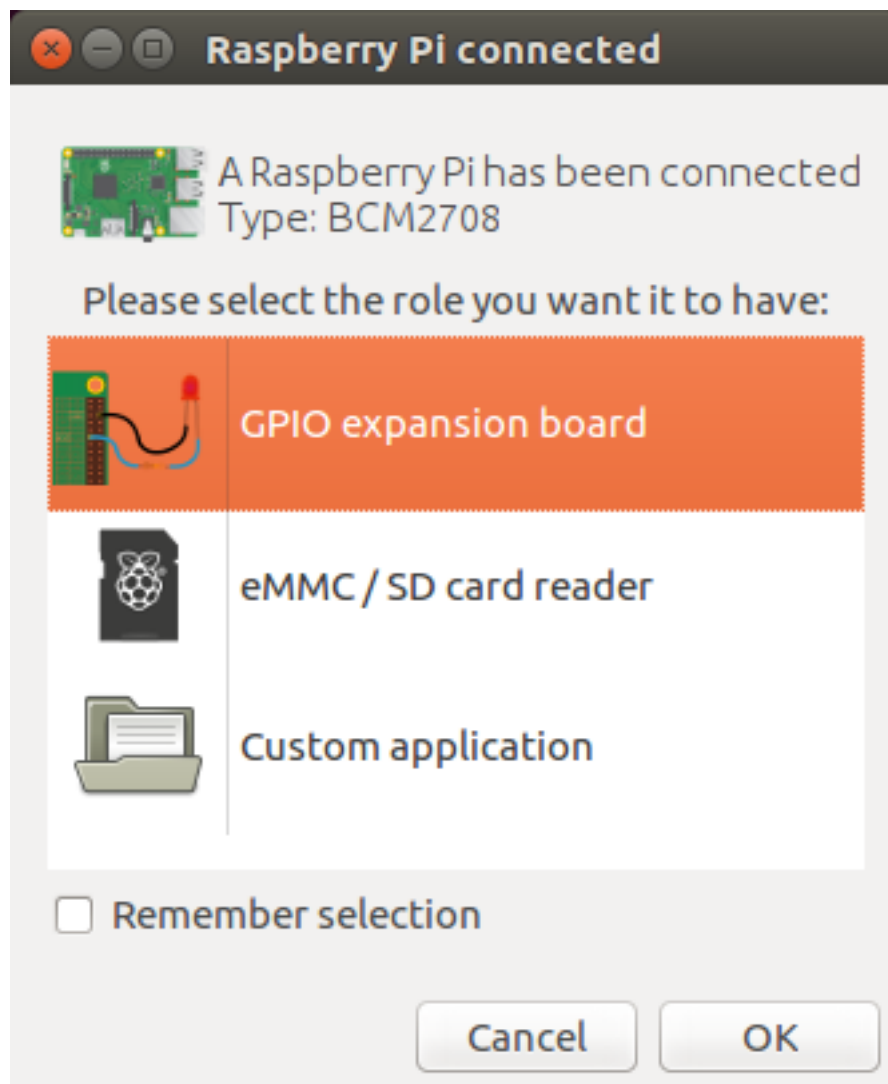
```
$ sudo pip3 uninstall gpiozero pigpio
```

3. Install the required packages from the PPA:

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

6.1.4 Access the GPIOs

Once your PC or Pi has the USB Boot GUI tool installed, connecting a Pi Zero will automatically launch a prompt to select a role for the device. Select “GPIO expansion board” and continue:



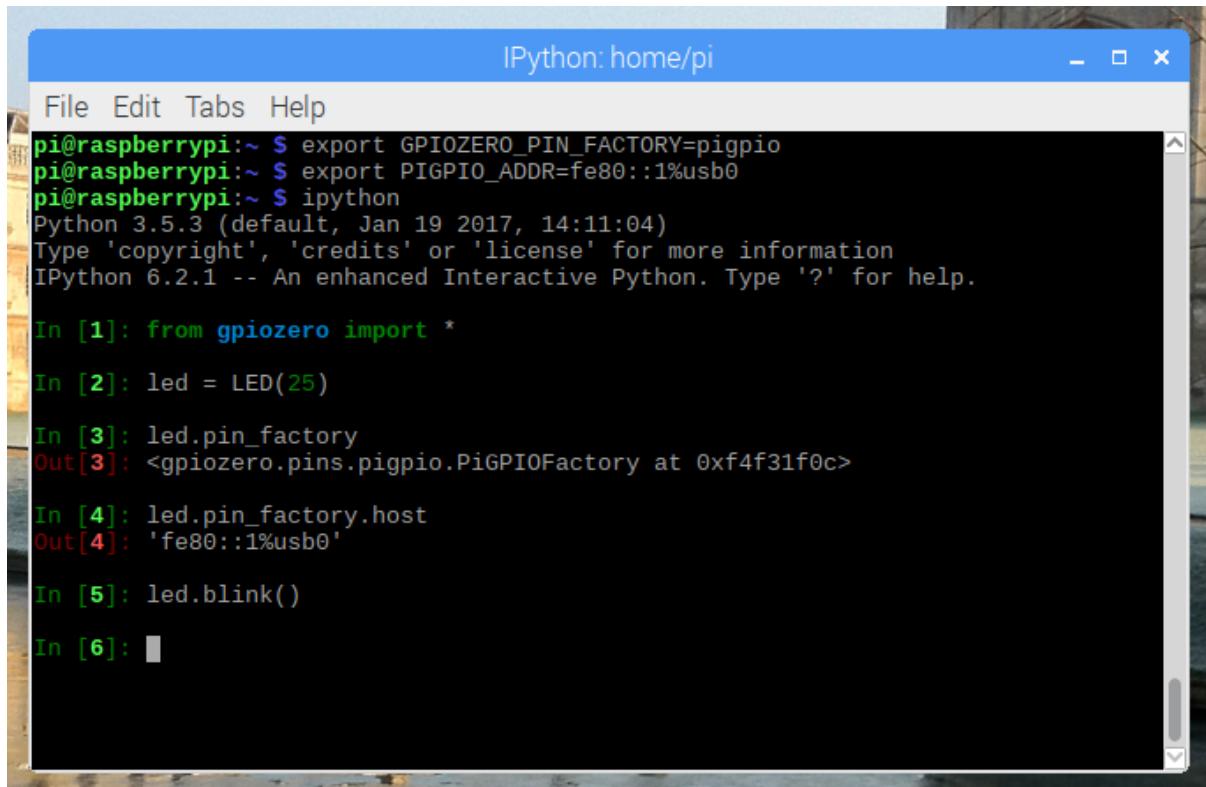
It will take 30 seconds or so to flash it, then the dialogue will disappear.

Raspberry Pi Desktop and Raspbian will name your Pi Zero connection `usb0`. On Ubuntu, this will likely be something else. You can ping it (be sure to use `ping6` as it's IPv6 only) using the address `fe80::1%` followed by the connection string. You can look this up using `ifconfig`.

Set the `GPIOWERO_PIN_FACTORY` and `PIGPIO_ADDR` environment variables on your PC so GPIO Zero connects to the “remote” Pi Zero:

```
$ export GPIOWERO_PIN_FACTORY=pigpio
$ export PIGPIO_ADDR=fe80::1%usb0
```

Now any GPIO Zero code you run on the PC will use the GPIOs of the attached Pi Zero:



The screenshot shows an IPython terminal window titled "IPython: home/pi". The terminal output is as follows:

```
File Edit Tabs Help
pi@raspberrypi:~ $ export GPIOWERO_PIN_FACTORY=pigpio
pi@raspberrypi:~ $ export PIGPIO_ADDR=fe80::1%usb0
pi@raspberrypi:~ $ ipython
Python 3.5.3 (default, Jan 19 2017, 14:11:04)
Type 'copyright', 'credits' or 'license' for more information
IPython 6.2.1 -- An enhanced Interactive Python. Type '?' for help.

In [1]: from gpiozero import *
In [2]: led = LED(25)
In [3]: led.pin_factory
Out[3]: <gpiozero.pins.pigpio.PiGPIOFactory at 0xf4f31f0c>
In [4]: led.pin_factory.host
Out[4]: 'fe80::1%usb0'
In [5]: led.blink()
In [6]:
```

Alternatively, you can set the pin factory in-line, as explained in [Configuring Remote GPIO](#) (page 35).

Read more on the GPIO expander in blog posts on raspberrypi.org³² and bennuttall.com³³.

6.2 Legacy method - SD card required

The legacy method requires the Pi Zero to have a Raspbian SD card inserted.

Start by creating a Raspbian (desktop or lite) SD card, and then configure the boot partition like so:

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.info³⁴ and learn.adafruit.com³⁵ 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 SSH into it using the hostname `raspberrypi.local`. SSH into the Pi Zero, install `pigpio` and run the `pigpio` daemon.

³² <https://www.raspberrypi.org/blog/gpio-expander/>

³³ <http://bennuttall.com/raspberry-pi-zero-gpio-expander/>

³⁴ <http://blog.gbaman.info/?p=791>

³⁵ <https://learn.adafruit.com/turning-your-raspberry-pi-zero-into-a-usb-gadget/ethernet-gadget>

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
```

CHAPTER 7

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)

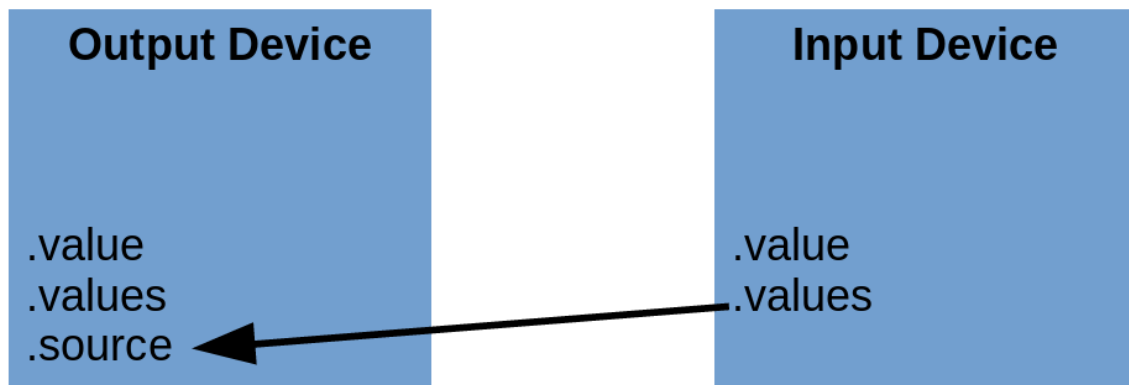
while True:
    led.value = button.value
    sleep(0.01)
```

Every device has a *value* (page 161) 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 162) property (a generator continuously yielding the device's current value). All output devices have a *source* (page 162) 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 162) 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 162) to the `values` (page 162) of another output device, to keep them matching:

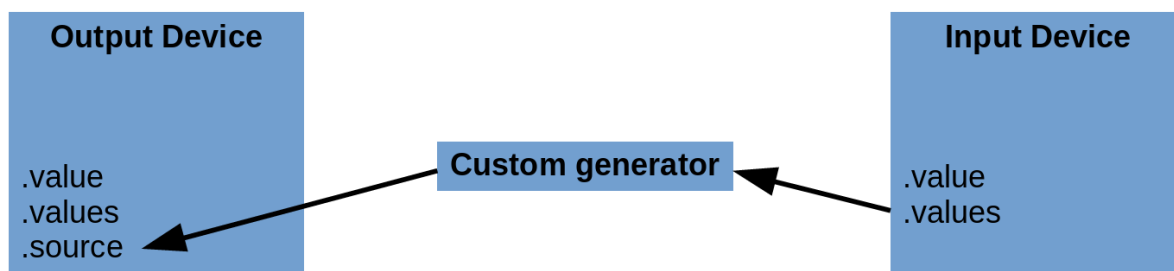
```
from gpiozero import LED, Button
from signal import pause

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

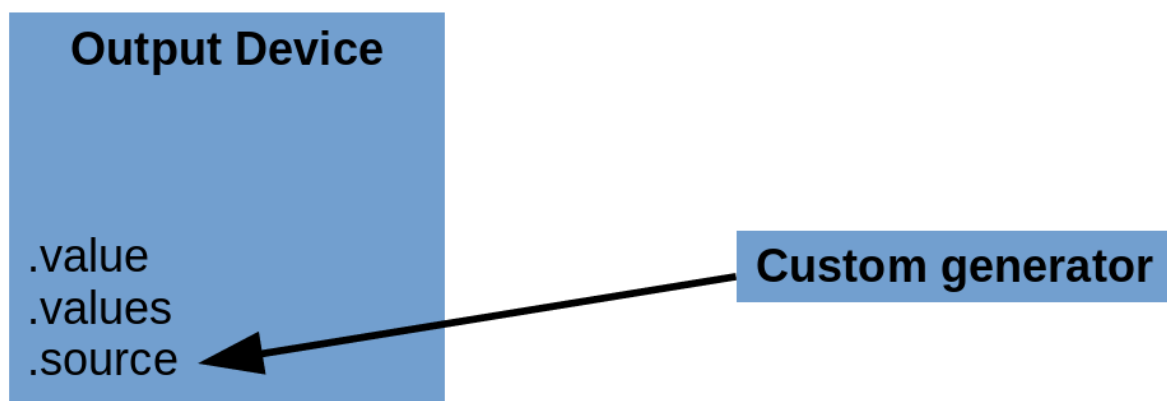
led = LED(4)
btn = Button(17)

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 162) 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()

```

7.1 Composite devices

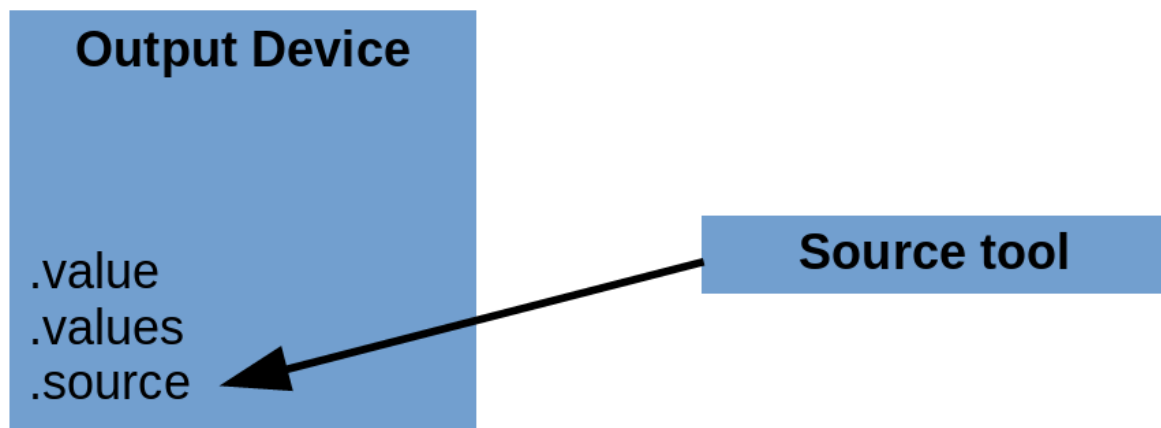
Most devices have a *value* (page 161) range between 0 and 1. Some have a range between -1 and 1 (e.g. *Motor* (page 93)). The *value* (page 161) of a composite device is a namedtuple of such values. For example, the *Robot* (page 136) class:

```
>>> 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
```

7.2 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 165).

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:

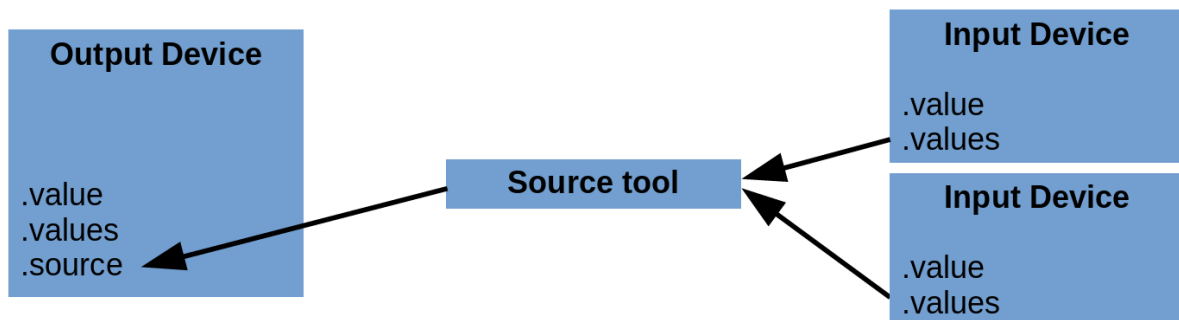
```
from gpiozero import Button, LED
from gpiozero.tools import negated
from signal import pause

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)³⁶ 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()
```

³⁶ https://en.wikipedia.org/wiki/AND_gate

CHAPTER 8

Command-line Tools

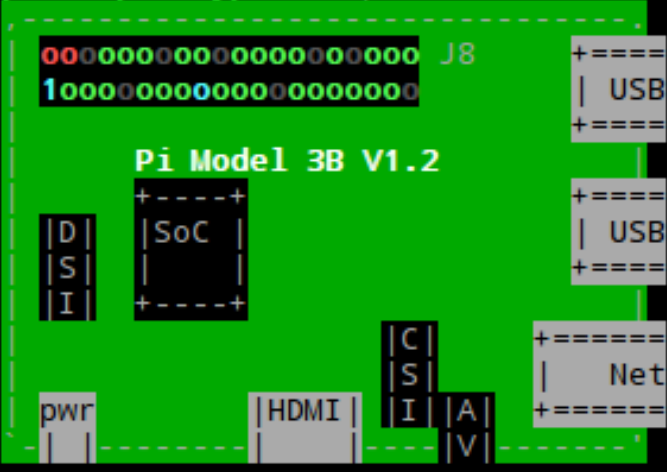
The `gpiozero` package contains a database of information about the various revisions of Raspberry Pi. This is queried by the **`pinout`** command-line tool to output details of the GPIO pins available.

8.1 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
  GPI02 (3) (4)  5V
  GPI03 (5) (6)  GND
  GPI04 (7) (8)  GPI014
  GND   (9) (10) GPI015
  GPI017 (11) (12) GPI018
  GPI027 (13) (14) GND
  GPI022 (15) (16) GPI023
  3V3   (17) (18) GPI024
  GPI010 (19) (20) GND
  GPI09  (21) (22) GPI025
  GPI011 (23) (24) GPI08
  GND    (25) (26) GPI07
  GPI00  (27) (28) GPI01
  GPI05  (29) (30) GND
  GPI06  (31) (32) GPI012
  GPI013 (33) (34) GND
  GPI019 (35) (36) GPI016
  GPI026 (37) (38) GPI020
  GND    (39) (40) GPI021

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

```

8.1.1 Synopsis

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

8.1.2 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.

8.1.3 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 60)
- m, --monochrome**
Force monochrome output. See also *--color* (page 60)

8.1.4 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
```

(continues on next page)

(continued from previous page)

```
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 60):

```
$ pinout --monochrome
```

Or forced to be `--color` (page 60), 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 60). The tool understands both old-style [revision codes](http://elinux.org/RPi_HardwareHistory)³⁷ (such as for the model B):

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

Or new-style [revision codes](http://elinux.org/RPi_HardwareHistory)³⁸ (such as for the Pi Zero W):

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

³⁷ http://elinux.org/RPi_HardwareHistory

³⁸ 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
```

8.1.5 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

9.1 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()`³⁹ 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()
```

³⁹ <https://docs.python.org/3.5/library/signal.html#signal.pause>

9.2 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).

9.3 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 177)). 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`⁴⁰ will be raised.

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

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

⁴⁰ <https://docs.python.org/3.5/library/exceptions.html#ImportError>

Refer to the `warnings`⁴¹ module documentation for more refined ways to filter out specific warning classes.

9.4 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`⁴².

⁴¹ <https://docs.python.org/3.5/library/warnings.html#module-warnings>

⁴² <https://pip.pypa.io/en/stable/installing/>

CHAPTER 10

Contributing

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

10.1 Suggestions

Please make suggestions for additional components or enhancements to the codebase by opening an [issue](#)⁴³ explaining your reasoning clearly.

10.2 Bugs

Please submit bug reports by opening an [issue](#)⁴⁴ explaining the problem clearly using code examples.

10.3 Documentation

The documentation source lives in the [docs](#)⁴⁵ folder. Contributions to the documentation are welcome but should be easy to read and understand.

10.4 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.

⁴³ <https://github.com/RPi-Distro/python-gpiozero/issues>

⁴⁴ <https://github.com/RPi-Distro/python-gpiozero/issues>

⁴⁵ <https://github.com/RPi-Distro/python-gpiozero/tree/master/docs>

10.5 Backwards compatibility

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

10.6 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/)⁴⁶.

⁴⁶ <http://legacy.python.org/dev/peps/pep-0373/>

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 87) and follow its heritage backward to *DigitalOutputDevice* (page 99). Follow that back to *OutputDevice* (page 102) 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).

11.1 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
```

11.2 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.

11.3 Test suite

If you wish to run the GPIO Zero test suite, follow the instructions in *Development installation* (page 71) 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 12

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.

12.1 Button

class `gpiozero.Button` (*pin*, *, *pull_up=True*, *bounce_time=None*, *hold_time=1*,
hold_repeat=False, *pin_factory=None*)

Extends *DigitalInputDevice* (page 82) 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 73) 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*⁴⁷) – The GPIO pin which the button is attached to. See *Pin Numbering* (page 3) for valid pin numbers.
- **pull_up** (*bool*⁴⁸) – 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.

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

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

- **bounce_time** (*float*⁴⁹) – 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*⁵⁰) – The length of time (in seconds) to wait after the button is pushed, until executing the *when_held* (page 74) handler. Defaults to 1.
- **hold_repeat** (*bool*⁵¹) – If `True`, the *when_held* (page 74) handler will be repeatedly executed as long as the device remains active, every *hold_time* seconds. If `False` (the default) the *when_held* (page 74) handler will be only be executed once per hold.
- **pin_factory** (*Factory* (page 180)) – See *API - Pins* (page 177) 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*⁵²) – 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*⁵³) – 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 74) event rather than when the device activated, in contrast to *active_time* (page 163). If the device is not currently held, this is `None`.

hold_repeat

If `True`, *when_held* (page 74) will be executed repeatedly with *hold_time* (page 74) 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 74) handler. If *hold_repeat* (page 74) is `True`, this is also the length of time between invocations of *when_held* (page 74).

is_held

When `True`, the device has been active for at least *hold_time* (page 74) 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 181) 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 74) seconds.

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

⁵⁰ <https://docs.python.org/3.5/library/functions.html#float>

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

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

⁵³ <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.

12.2 Line Sensor (TRCT5000)

class `gpiozero.LineSensor` (*pin*, *, *queue_len*=5, *sample_rate*=100, *threshold*=0.5, *partial*=False, *pin_factory*=None)

Extends `SmoothedInputDevice` (page 82) and represents a single pin line sensor like the TCRT5000 infra-red proximity sensor found in the [CamJam #3 EduKit](#)⁵⁴.

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*⁵⁵) – The GPIO pin which the sensor is attached to. See [Pin Numbering](#) (page 3) for valid pin numbers.
- **queue_len** (*int*⁵⁶) – The length of the queue used to store values read from the sensor. This defaults to 5.
- **sample_rate** (*float*⁵⁷) – The number of values to read from the device (and append to the internal queue) per second. Defaults to 100.

⁵⁴ http://camjam.me/?page_id=1035

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

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

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

- **threshold** (*float*⁵⁸) – 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 84) property, and all appropriate events will be fired.
- **partial** (*bool*⁵⁹) – When `False` (the default), the object will not return a value for *is_active* (page 84) 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 180)) – See *API - Pins* (page 177) 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*⁶⁰) – 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*⁶¹) – 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 181) 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.

12.3 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 82) and represents a passive infra-red (PIR) motion sensor like the sort found in the *CamJam #2 EduKit*⁶².

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:

⁵⁸ <https://docs.python.org/3.5/library/functions.html#float>

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

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

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

⁶² http://camjam.me/?page_id=623


```
from gpiozero import MotionSensor

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

Parameters

- **pin** (*int*⁶³) – The GPIO pin which the sensor is attached to. See *Pin Numbering* (page 3) for valid pin numbers.
- **queue_len** (*int*⁶⁴) – 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*⁶⁵) – The number of values to read from the device (and append to the internal queue) per second. Defaults to 100.
- **threshold** (*float*⁶⁶) – 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 84) property, and all appropriate events will be fired.
- **partial** (*bool*⁶⁷) – When *False* (the default), the object will not return a value for *is_active* (page 84) until the internal queue has filled with values. Only set this to *True* if you require values immediately after object construction.
- **pull_up** (*bool*⁶⁸) – If *False* (the default), the GPIO pin will be pulled low by default. If *True*, the GPIO pin will be pulled high by the sensor.
- **pin_factory** (*Factory* (page 180)) – See *API - Pins* (page 177) 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*⁶⁹) – 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*⁷⁰) – 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 181) 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.

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

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

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

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

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

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

⁶⁹ <https://docs.python.org/3.5/library/functions.html#float>

⁷⁰ <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_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.

12.4 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 82) and represents a light dependent resistor (LDR).

Connect one leg of the LDR to the 3V3 pin; connect one leg of a 1 μ 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*⁷¹) – The GPIO pin which the sensor is attached to. See *Pin Numbering* (page 3) for valid pin numbers.
- **queue_len** (*int*⁷²) – The length of the queue used to store values read from the circuit. This defaults to 5.
- **charge_time_limit** (*float*⁷³) – 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 μ F capacitor coupled with the LDR from the *CamJam #2 EduKit*⁷⁴. You may need to adjust this value for different valued capacitors or LDRs.
- **threshold** (*float*⁷⁵) – 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*⁷⁶) – When `False` (the default), the object will not return a value for *is_active* (page 84) 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 180)) – See *API - Pins* (page 177) for more information (this is an advanced feature which most users can ignore).

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

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

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

⁷⁴ http://camjam.me/?page_id=623

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

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

wait_for_dark (*timeout=None*)

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

Parameters **timeout** (*float*⁷⁷) – 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*⁷⁸) – 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 181) 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.

12.5 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 82) and represents an HC-SR04 ultrasonic distance sensor, as found in the `CamJam #3 EduKit`⁷⁹.

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 one end of a 330Ω resistor to the ECHO pin of the sensor.
4. Connect one end of a 470Ω resistor to the GND pin of the sensor.
5. Connect the free ends of both resistors to another GPIO pin. This forms the required `voltage divider`⁸⁰.
6. Finally, connect the VCC pin of the sensor to a 5V pin on the Pi.

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

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

⁷⁹ http://camjam.me/?page_id=1035

⁸⁰ https://en.wikipedia.org/wiki/Voltage_divider

Note: If you do not have the precise values of resistor specified above, don't worry! What matters is the *ratio* of the resistors to each other.

You also don't need to be absolutely precise; the [voltage divider](#)⁸¹ given above will actually output ~3V (rather than 3.3V). A simple 2:3 ratio will give 3.333V which implies you can take three resistors of equal value, use one of them instead of the 330Ω resistor, and two of them in series instead of the 470Ω resistor.

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

```
from gpiozero import DistanceSensor
from time import sleep

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

Parameters

- **echo** (*int*⁸²) – The GPIO pin which the ECHO pin is attached to. See [Pin Numbering](#) (page 3) for valid pin numbers.
- **trigger** (*int*⁸³) – The GPIO pin which the TRIG pin is attached to. See [Pin Numbering](#) (page 3) for valid pin numbers.
- **queue_len** (*int*⁸⁴) – The length of the queue used to store values read from the sensor. This defaults to 30.
- **max_distance** (*float*⁸⁵) – 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*⁸⁶) – 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*⁸⁷) – When `False` (the default), the object will not return a value for `is_active` (page 84) 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 180)) – See [API - Pins](#) (page 177) 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*⁸⁸) – 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*⁸⁹) – Number of seconds to wait before proceeding. If this is `None` (the default), then wait indefinitely until the device is active.

⁸¹ https://en.wikipedia.org/wiki/Voltage_divider

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

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

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

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

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

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

⁸⁸ <https://docs.python.org/3.5/library/functions.html#float>

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

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 81).

echo

Returns the `Pin` (page 181) 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 80) is equal to `max_distance` (page 81), `value` will be 1.

threshold_distance

The distance, measured in meters, that will trigger the `when_in_range` (page 81) and `when_out_of_range` (page 81) events when crossed. This is simply a meter-scaled variant of the usual `threshold` attribute.

trigger

Returns the `Pin` (page 181) 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

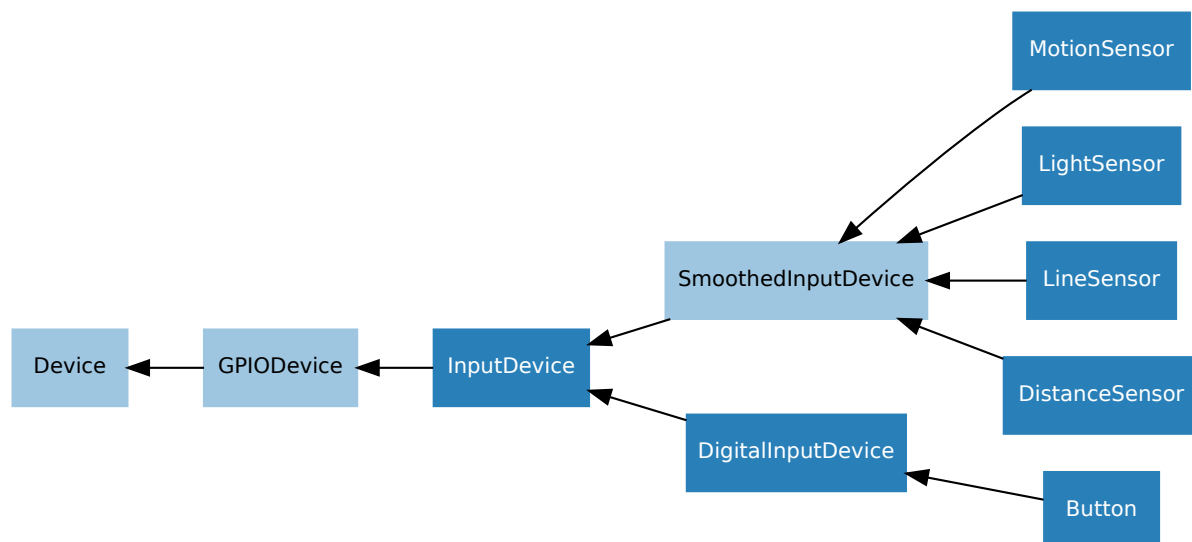
The function to run when the device changes state from inactive to active.

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.

12.6 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.

12.7 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 84) 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*⁹⁰) – 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 180)) – See *API - Pins* (page 177) for more information (this is an advanced feature which most users can ignore).

12.8 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 average of a queue of historical values.

This class extends `InputDevice` (page 84) with a queue which is filled by a background thread which continually polls the state of the underlying device. The average (a configurable function) of the values in the queue is compared to a threshold which is used to determine the state of the `is_active` (page 84) 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.

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

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*⁹¹) – The value above which the device will be considered “on”.
- **queue_len** (*int*⁹²) – The length of the internal queue which is filled by the background thread.
- **sample_wait** (*float*⁹³) – 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*⁹⁴) – If `False` (the default), attempts to read the state of the device (from the `is_active` (page 84) 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.
- **average** – The function used to average the values in the internal queue. This defaults to `statistics.median()`⁹⁵ which is a good selection for discarding outliers from jittery sensors. The function specific must accept a sequence of numbers and return a single number.
- **pin_factory** (`Factory` (page 180)) – See *API - Pins* (page 177) 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:

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

Device (page 161) descendants can also be used as context managers using the `with`⁹⁶ statement. For example:

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

(continues on next page)

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

⁹² <https://docs.python.org/3.5/library/functions.html#int>

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

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

⁹⁵ <https://docs.python.org/3.5/library/statistics.html#statistics.median>

⁹⁶ https://docs.python.org/3.5/reference/compound_stmts.html#with

(continued from previous page)

```
...     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 84) or `is_active` (page 84) 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 84) exceeds this amount, then `is_active` (page 84) will return `True`.

value

Returns the mean of the values in the internal queue. This is compared to `threshold` (page 84) to determine whether `is_active` (page 84) is `True`.

12.9 InputDevice

class gpiozero.InputDevice (pin, *, pull_up=False, pin_factory=None)

Represents a generic GPIO input device.

This class extends `GPIODevice` (page 84) 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 84) setting.

Parameters

- **pin** (`int`⁹⁷) – The GPIO pin (in Broadcom numbering) that the device is connected to. If this is `None` a `GPIODeviceError` (page 192) will be raised.
- **pull_up** (`bool`⁹⁸) – 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 180)) – See *API - Pins* (page 177) 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.

12.10 GPIODevice

class gpiozero.GPIODevice (pin, pin_factory=None)

Extends `Device` (page 161). 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 85)).

Parameters **pin** (`int`⁹⁹) – The GPIO pin (in BCM numbering) that the device is connected to. If this is `None`, `GPIOPinMissing` (page 192) will be raised. If the pin is already in use by another device, `GPIOPinInUse` (page 192) will be raised.

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

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

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

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 161) descendants can also be used as context managers using the `with`¹⁰⁰ 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 84) method). Once a device is closed you can no longer use any other methods or properties to control or query the device.

pin

The *Pin* (page 181) that the device is connected to. This will be `None` if the device has been closed (see the `close()` (page 84) method). When dealing with GPIO pins, query `pin.number` to discover the GPIO pin (in BCM numbering) that the device is connected to.

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.

¹⁰⁰ https://docs.python.org/3.5/reference/compound_stmts.html#with

CHAPTER 13

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.

13.1 LED

class `gpiozero.LED` (*pin*, *, *active_high=True*, *initial_value=False*, *pin_factory=None*)

Extends *DigitalOutputDevice* (page 99) 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*¹⁰¹) – The GPIO pin which the LED is attached to. See *Pin Numbering* (page 3) for valid pin numbers.
- **active_high** (*bool*¹⁰²) – 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).

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

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

- **initial_value** (*bool*¹⁰³) – 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 180)) – See *API - Pins* (page 177) 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*¹⁰⁴) – Number of seconds on. Defaults to 1 second.
- **off_time** (*float*¹⁰⁵) – Number of seconds off. Defaults to 1 second.
- **n** (*int*¹⁰⁶) – Number of times to blink; `None` (the default) means forever.
- **background** (*bool*¹⁰⁷) – 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 181) 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.

13.2 PWMLED

class `gpiozero.PWMLED` (*pin*, *, *active_high=True*, *initial_value=0*, *frequency=100*,
pin_factory=None)

Extends *PWMOutputDevice* (page 100) 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*¹⁰⁸) – The GPIO pin which the LED is attached to. See *Pin Numbering* (page 3) for valid pin numbers.
- **active_high** (*bool*¹⁰⁹) – If `True` (the default), the `on()` (page 89) method will set the GPIO to HIGH. If `False`, the `on()` (page 89) method will set the GPIO to LOW (the `off()` (page 89) method always does the opposite).

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

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

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

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

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

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

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

- **initial_value** (*float*¹¹⁰) – 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*¹¹¹) – The frequency (in Hz) of pulses emitted to drive the LED. Defaults to 100Hz.
- **pin_factory** (`Factory` (page 180)) – See *API - Pins* (page 177) 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*¹¹²) – Number of seconds on. Defaults to 1 second.
- **off_time** (*float*¹¹³) – Number of seconds off. Defaults to 1 second.
- **fade_in_time** (*float*¹¹⁴) – Number of seconds to spend fading in. Defaults to 0.
- **fade_out_time** (*float*¹¹⁵) – Number of seconds to spend fading out. Defaults to 0.
- **n** (*int*¹¹⁶) – Number of times to blink; `None` (the default) means forever.
- **background** (*bool*¹¹⁷) – 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*¹¹⁸) – Number of seconds to spend fading in. Defaults to 1.
- **fade_out_time** (*float*¹¹⁹) – Number of seconds to spend fading out. Defaults to 1.
- **n** (*int*¹²⁰) – Number of times to pulse; `None` (the default) means forever.
- **background** (*bool*¹²¹) – 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 90) is 0.0), this changes it to “fully” on (*value* (page 90) is 1.0). If the device has a duty cycle (*value* (page 90)) of 0.1, this will toggle it to 0.9, and so on.

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

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

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

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

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

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

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

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

¹¹⁸ <https://docs.python.org/3.5/library/functions.html#float>

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

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

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

is_lit

Returns `True` if the device is currently active (*value* (page 90) is non-zero) and `False` otherwise.

pin

The *Pin* (page 181) 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.

13.3 RGBLED

class `gpiozero.RGBLED` (*red, green, blue, *, active_high=True, initial_value=(0, 0, 0), pwm=True, pin_factory=None*)

Extends *Device* (page 161) 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*¹²²) – The GPIO pin that controls the red component of the RGB LED.
- **green** (*int*¹²³) – The GPIO pin that controls the green component of the RGB LED.
- **blue** (*int*¹²⁴) – The GPIO pin that controls the blue component of the RGB LED.
- **active_high** (*bool*¹²⁵) – 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*¹²⁶) – The initial color for the RGB LED. Defaults to black `(0, 0, 0)`.
- **pwm** (*bool*¹²⁷) – If `True` (the default), construct *PWMLED* (page 88) instances for each component of the RGBLED. If `False`, construct regular *LED* (page 87) instances, which prevents smooth color graduations.
- **pin_factory** (*Factory* (page 180)) – See *API - Pins* (page 177) 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

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

¹²³ <https://docs.python.org/3.5/library/functions.html#int>

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

¹²⁵ <https://docs.python.org/3.5/library/functions.html#bool>

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

¹²⁷ <https://docs.python.org/3.5/library/functions.html#bool>

- **on_time** (*float*¹²⁸) – Number of seconds on. Defaults to 1 second.
- **off_time** (*float*¹²⁹) – Number of seconds off. Defaults to 1 second.
- **fade_in_time** (*float*¹³⁰) – Number of seconds to spend fading in. Defaults to 0. Must be 0 if `pwm` was `False` when the class was constructed (`ValueError`¹³¹ will be raised if not).
- **fade_out_time** (*float*¹³²) – Number of seconds to spend fading out. Defaults to 0. Must be 0 if `pwm` was `False` when the class was constructed (`ValueError`¹³³ will be raised if not).
- **on_color** (*tuple*¹³⁴) – The color to use when the LED is “on”. Defaults to white.
- **off_color** (*tuple*¹³⁵) – The color to use when the LED is “off”. Defaults to black.
- **n** (*int*¹³⁶) – Number of times to blink; `None` (the default) means forever.
- **background** (*bool*¹³⁷) – 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*¹³⁸) – Number of seconds to spend fading in. Defaults to 1.
- **fade_out_time** (*float*¹³⁹) – Number of seconds to spend fading out. Defaults to 1.
- **on_color** (*tuple*¹⁴⁰) – The color to use when the LED is “on”. Defaults to white.
- **off_color** (*tuple*¹⁴¹) – The color to use when the LED is “off”. Defaults to black.
- **n** (*int*¹⁴²) – Number of times to pulse; `None` (the default) means forever.
- **background** (*bool*¹⁴³) – 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.

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

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

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

¹³¹ <https://docs.python.org/3.5/library/exceptions.html#ValueError>

¹³² <https://docs.python.org/3.5/library/functions.html#float>

¹³³ <https://docs.python.org/3.5/library/exceptions.html#ValueError>

¹³⁴ <https://docs.python.org/3.5/library/stdtypes.html#tuple>

¹³⁵ <https://docs.python.org/3.5/library/stdtypes.html#tuple>

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

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

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

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

¹⁴⁰ <https://docs.python.org/3.5/library/stdtypes.html#tuple>

¹⁴¹ <https://docs.python.org/3.5/library/stdtypes.html#tuple>

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

¹⁴³ <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.

13.4 Buzzer

class `gpiozero.Buzzer` (*pin*, *, *active_high*=`True`, *initial_value*=`False`, *pin_factory*=`None`)

Extends *DigitalOutputDevice* (page 99) 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*¹⁴⁴) – The GPIO pin which the buzzer is attached to. See *Pin Numbering* (page 3) for valid pin numbers.
- **active_high** (*bool*¹⁴⁵) – 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*¹⁴⁶) – 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 180)) – See *API - Pins* (page 177) 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*¹⁴⁷) – Number of seconds on. Defaults to 1 second.
- **off_time** (*float*¹⁴⁸) – Number of seconds off. Defaults to 1 second.
- **n** (*int*¹⁴⁹) – Number of times to blink; `None` (the default) means forever.
- **background** (*bool*¹⁵⁰) – 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).

¹⁴⁴ <https://docs.python.org/3.5/library/functions.html#int>

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

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

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

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

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

¹⁵⁰ <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 181) 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.

13.5 Motor

class `gpiozero.Motor` (*forward, backward, *, pwm=True, pin_factory=None*)

Extends `CompositeDevice` (page 153) and represents a generic motor connected to a bi-directional motor driver circuit (i.e. an `H-bridge`¹⁵¹).

Attach an `H-bridge`¹⁵² 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`¹⁵³) – The GPIO pin that the forward input of the motor driver chip is connected to.
- **backward** (`int`¹⁵⁴) – The GPIO pin that the backward input of the motor driver chip is connected to.
- **pwm** (`bool`¹⁵⁵) – If `True` (the default), construct `PWMOutputDevice` (page 100) instances for the motor controller pins, allowing both direction and variable speed control. If `False`, construct `DigitalOutputDevice` (page 99) instances, allowing only direction control.
- **pin_factory** (`Factory` (page 180)) – See *API - Pins* (page 177) for more information (this is an advanced feature which most users can ignore).

backward (*speed=1*)

Drive the motor backwards.

Parameters **speed** (`float`¹⁵⁶) – 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).

¹⁵¹ https://en.wikipedia.org/wiki/H_bridge

¹⁵² https://en.wikipedia.org/wiki/H_bridge

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

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

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

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

forward (*speed=1*)

Drive the motor forwards.

Parameters **speed** (*float*¹⁵⁷) – 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).

reverse ()

Reverse the current direction of the motor. If the motor is currently idle this does nothing. Otherwise, the motor’s direction will be reversed at the current speed.

stop ()

Stop the motor.

13.6 PhaseEnableMotor

class `gpiozero.PhaseEnableMotor` (*phase, enable, *, pwm=True, pin_factory=None*)

Extends *CompositeDevice* (page 153) and represents a generic motor connected to a Phase/Enable motor driver circuit; the phase of the driver controls whether the motor turns forwards or backwards, while enable controls the speed with PWM.

The following code will make the motor turn “forwards”:

```
from gpiozero import PhaseEnableMotor
motor = PhaseEnableMotor(12, 5)
motor.forward()
```

Parameters

- **phase** (*int*¹⁵⁸) – The GPIO pin that the phase (direction) input of the motor driver chip is connected to.
- **enable** (*int*¹⁵⁹) – The GPIO pin that the enable (speed) input of the motor driver chip is connected to.
- **pwm** (*bool*¹⁶⁰) – If `True` (the default), construct *PWMOutputDevice* (page 100) instances for the motor controller pins, allowing both direction and variable speed control. If `False`, construct *DigitalOutputDevice* (page 99) instances, allowing only direction control.
- **pin_factory** (*Factory* (page 180)) – See *API - Pins* (page 177) for more information (this is an advanced feature which most users can ignore).

backward (*speed=1*)

Drive the motor backwards.

Parameters **speed** (*float*¹⁶¹) – The speed at which the motor should turn. Can be any value between 0 (stopped) and the default 1 (maximum speed).

forward (*speed=1*)

Drive the motor forwards.

Parameters **speed** (*float*¹⁶²) – The speed at which the motor should turn. Can be any value between 0 (stopped) and the default 1 (maximum speed).

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

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

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

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

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

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

reverse()

Reverse the current direction of the motor. If the motor is currently idle this does nothing. Otherwise, the motor's direction will be reversed at the current speed.

stop()

Stop the motor.

13.7 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 153) 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](#)¹⁶³) – The GPIO pin which the device is attached to. See [Pin Numbering](#) (page 3) for valid pin numbers.
- **initial_value** ([float](#)¹⁶⁴) – 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 96)).
- **min_pulse_width** ([float](#)¹⁶⁵) – The pulse width corresponding to the servo's minimum position. This defaults to 1ms.
- **max_pulse_width** ([float](#)¹⁶⁶) – The pulse width corresponding to the servo's maximum position. This defaults to 2ms.
- **frame_width** ([float](#)¹⁶⁷) – 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 180)) – See [API - Pins](#) (page 177) 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 96) to None.

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

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

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

¹⁶⁶ <https://docs.python.org/3.5/library/functions.html#float>

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

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.

closed
Returns `True` if the device is closed (see the `close()` method). Once a device is closed you can no longer use any other methods or properties to control or query the device.

frame_width
The time between control pulses, measured in seconds.

is_active
Returns `True` if the device is currently active and `False` otherwise. This property is usually derived from `value` (page 96). Unlike `value` (page 96), this is *always* a boolean.

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 96).

source_delay
The delay (measured in seconds) in the loop used to read values from `source` (page 96). 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`.

13.8 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 95) 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 95) 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 96) 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*¹⁶⁸) – The GPIO pin which the device is attached to. See *Pin Numbering* (page 3) for valid pin numbers.
- **initial_angle** (*float*¹⁶⁹) – 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 98)).
- **min_angle** (*float*¹⁷⁰) – 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*¹⁷¹) – 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*¹⁷²) – The pulse width corresponding to the servo’s minimum position. This defaults to 1ms.
- **max_pulse_width** (*float*¹⁷³) – The pulse width corresponding to the servo’s maximum position. This defaults to 2ms.
- **frame_width** (*float*¹⁷⁴) – 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 180)) – See *API - Pins* (page 177) 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 98) to None.

max()

Set the servo to its maximum position.

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

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

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

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

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

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

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

mid()

Set the servo to its mid-point position.

min()

Set the servo to its minimum position.

angle

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

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

closed

Returns `True` if the device is closed (see the `close()` method). Once a device is closed you can no longer use any other methods or properties to control or query the device.

frame_width

The time between control pulses, measured in seconds.

is_active

Returns `True` if the device is currently active and `False` otherwise. This property is usually derived from *value* (page 98). Unlike *value* (page 98), this is *always* a boolean.

max_angle

The maximum angle that the servo will rotate to when *max()* (page 97) 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 98) 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 98).

source_delay

The delay (measured in seconds) in the loop used to read values from *source* (page 98). Defaults to 0.01 seconds which is generally sufficient to keep CPU usage to a minimum while providing adequate responsiveness.

value

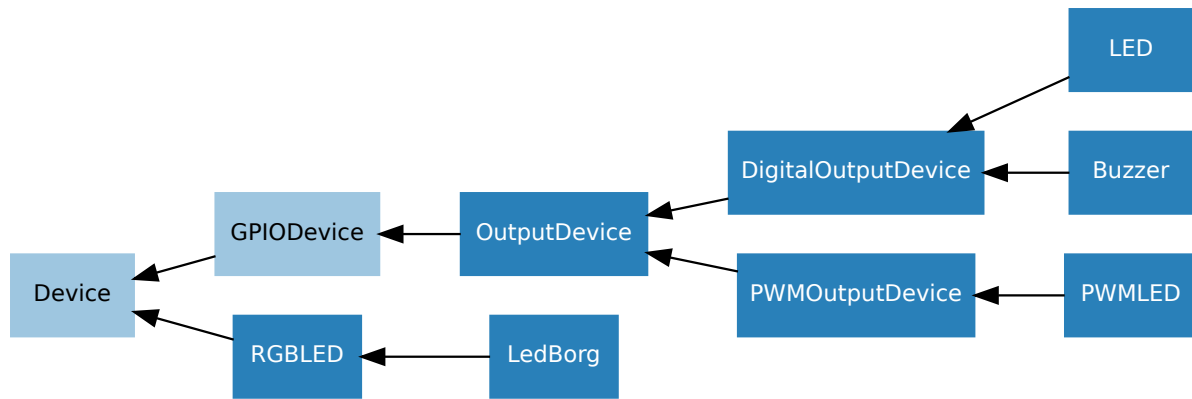
Represents the position of the servo as a value between -1 (the minimum position) and +1 (the maximum position). This can also be the special value `None` indicating that the servo is currently “uncontrolled”, i.e. that no control signal is being sent. Typically this means the servo’s position remains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

values

An infinite iterator of values read from *value*.

13.9 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.

13.10 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 102) with a `blink()` (page 99) 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*¹⁷⁵) – Number of seconds on. Defaults to 1 second.
- **off_time** (*float*¹⁷⁶) – Number of seconds off. Defaults to 1 second.
- **n** (*int*¹⁷⁷) – Number of times to blink; `None` (the default) means forever.
- **background** (*bool*¹⁷⁸) – 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)
```

(continues on next page)

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

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

¹⁷⁷ <https://docs.python.org/3.5/library/functions.html#int>

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

(continued from previous page)

```
>>> bz.on()
>>> bz.off()
>>> bz.close()
>>> led = LED(16)
>>> led.blink()
```

Device (page 161) descendants can also be used as context managers using the `with`¹⁷⁹ 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.

value

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

13.11 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*¹⁸⁰) – The GPIO pin which the device is attached to. See *Pin Numbering* (page 3) for valid pin numbers.
- **active_high** (*bool*¹⁸¹) – If `True` (the default), the `on()` (page 101) method will set the GPIO to HIGH. If `False`, the `on()` (page 101) method will set the GPIO to LOW (the `off()` (page 101) method always does the opposite).
- **initial_value** (*float*¹⁸²) – 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*¹⁸³) – The frequency (in Hz) of pulses emitted to drive the device. Defaults to 100Hz.
- **pin_factory** (*Factory* (page 180)) – See *API - Pins* (page 177) 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

¹⁷⁹ https://docs.python.org/3.5/reference/compound_stmts.html#with

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

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

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

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

- **on_time** (*float*¹⁸⁴) – Number of seconds on. Defaults to 1 second.
- **off_time** (*float*¹⁸⁵) – Number of seconds off. Defaults to 1 second.
- **fade_in_time** (*float*¹⁸⁶) – Number of seconds to spend fading in. Defaults to 0.
- **fade_out_time** (*float*¹⁸⁷) – Number of seconds to spend fading out. Defaults to 0.
- **n** (*int*¹⁸⁸) – Number of times to blink; `None` (the default) means forever.
- **background** (*bool*¹⁸⁹) – 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 161) descendents can also be used as context managers using the `with`¹⁹⁰ 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

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

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

¹⁸⁶ <https://docs.python.org/3.5/library/functions.html#float>

¹⁸⁷ <https://docs.python.org/3.5/library/functions.html#float>

¹⁸⁸ <https://docs.python.org/3.5/library/functions.html#int>

¹⁸⁹ <https://docs.python.org/3.5/library/functions.html#bool>

¹⁹⁰ https://docs.python.org/3.5/reference/compound_stmts.html#with

- **fade_in_time** (*float*¹⁹¹) – Number of seconds to spend fading in. Defaults to 1.
- **fade_out_time** (*float*¹⁹²) – Number of seconds to spend fading out. Defaults to 1.
- **n** (*int*¹⁹³) – Number of times to pulse; `None` (the default) means forever.
- **background** (*bool*¹⁹⁴) – 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 102) is 0.0), this changes it to “fully” on (*value* (page 102) is 1.0). If the device has a duty cycle (*value* (page 102)) of 0.1, this will toggle it to 0.9, and so on.

frequency

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

is_active

Returns `True` if the device is currently active (*value* (page 102) 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.

13.12 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 84) to add facilities common to GPIO output devices: an *on()* (page 102) method to switch the device on, a corresponding *off()* (page 102) method, and a *toggle()* (page 102) method.

Parameters

- **pin** (*int*¹⁹⁵) – The GPIO pin (in BCM numbering) that the device is connected to. If this is `None` a *GPIOPinMissing* (page 192) will be raised.
- **active_high** (*bool*¹⁹⁶) – If `True` (the default), the *on()* (page 102) method will set the GPIO to HIGH. If `False`, the *on()* (page 102) method will set the GPIO to LOW (the *off()* (page 102) method always does the opposite).
- **initial_value** (*bool*¹⁹⁷) – 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 180)) – See *API - Pins* (page 177) for more information (this is an advanced feature which most users can ignore).

off()

Turns the device off.

on()

Turns the device on.

¹⁹¹ <https://docs.python.org/3.5/library/functions.html#float>

¹⁹² <https://docs.python.org/3.5/library/functions.html#float>

¹⁹³ <https://docs.python.org/3.5/library/functions.html#int>

¹⁹⁴ <https://docs.python.org/3.5/library/functions.html#bool>

¹⁹⁵ <https://docs.python.org/3.5/library/functions.html#int>

¹⁹⁶ <https://docs.python.org/3.5/library/functions.html#bool>

¹⁹⁷ <https://docs.python.org/3.5/library/functions.html#bool>

`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 103) property is `True` when the device's pin is high. When `False` the `value` (page 103) 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 103) (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.

13.13 GPIODevice

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

Extends [Device](#) (page 161). Represents a generic GPIO device and provides the services common to all single-pin GPIO devices (like ensuring two GPIO devices do no share a [pin](#) (page 85)).

Parameters `pin` (*int*¹⁹⁸) – The GPIO pin (in BCM numbering) that the device is connected to. If this is `None`, *GPIOPinMissing* (page 192) will be raised. If the pin is already in use by another device, *GPIOPinInUse* (page 192) 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 161) descendants can also be used as context managers using the `with`¹⁹⁹ statement. For example:

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

¹⁹⁸ <https://docs.python.org/3.5/library/functions.html#int>

199 https://docs.python.org/3.5/reference/compound_stmts.html#with

closed

Returns `True` if the device is closed (see the `close()` (page 84) method). Once a device is closed you can no longer use any other methods or properties to control or query the device.

pin

The `Pin` (page 181) that the device is connected to. This will be `None` if the device has been closed (see the `close()` (page 84) method). When dealing with GPIO pins, query `pin.number` to discover the GPIO pin (in BCM numbering) that the device is connected to.

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.

SPI stands for [Serial Peripheral Interface](#)²⁰⁰ 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.

14.1 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 192) being raised. However, you can omit any arguments from either scheme. The defaults are:

²⁰⁰ 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 177) 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.

14.2 Analog to Digital Converters (ADC)

class gpiozero.**MCP3001** (*max_voltage=3.3, **spi_args*)

The **MCP3001**²⁰¹ 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, max_voltage=3.3, **spi_args*)

The **MCP3002**²⁰² 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 107) 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, max_voltage=3.3, **spi_args*)

The **MCP3004**²⁰³ 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.

²⁰¹ <http://www.farnell.com/datasheets/630400.pdf>

²⁰² <http://www.farnell.com/datasheets/1599363.pdf>

²⁰³ <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 107) 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, max_voltage=3.3, **spi_args*)

The [MCP3008](#)²⁰⁴ 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 107) 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` (*max_voltage=3.3, **spi_args*)

The [MCP3201](#)²⁰⁵ 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, max_voltage=3.3, **spi_args*)

The [MCP3202](#)²⁰⁶ 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 107) 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, max_voltage=3.3, **spi_args*)

The [MCP3204](#)²⁰⁷ is a 12-bit analog to digital converter with 4 channels (0-3).

²⁰⁴ <http://www.farnell.com/datasheets/808965.pdf>

²⁰⁵ <http://www.farnell.com/datasheets/1669366.pdf>

²⁰⁶ <http://www.farnell.com/datasheets/1669376.pdf>

²⁰⁷ <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 107) 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, max_voltage=3.3, **spi_args*)

The [MCP3208](#)²⁰⁸ 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 107) 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** (*max_voltage=3.3, **spi_args*)

The [MCP3301](#)²⁰⁹ 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, max_voltage=3.3, **spi_args*)

The [MCP3302](#)²¹⁰ 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).

²⁰⁸ <http://www.farnell.com/datasheets/808967.pdf>

²⁰⁹ <http://www.farnell.com/datasheets/1669397.pdf>

²¹⁰ <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 109) 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, max_voltage=3.3, **spi_args*)

The [MCP3304](#)²¹¹ 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 109) in differential mode, channel 0 is read relative to channel 1).

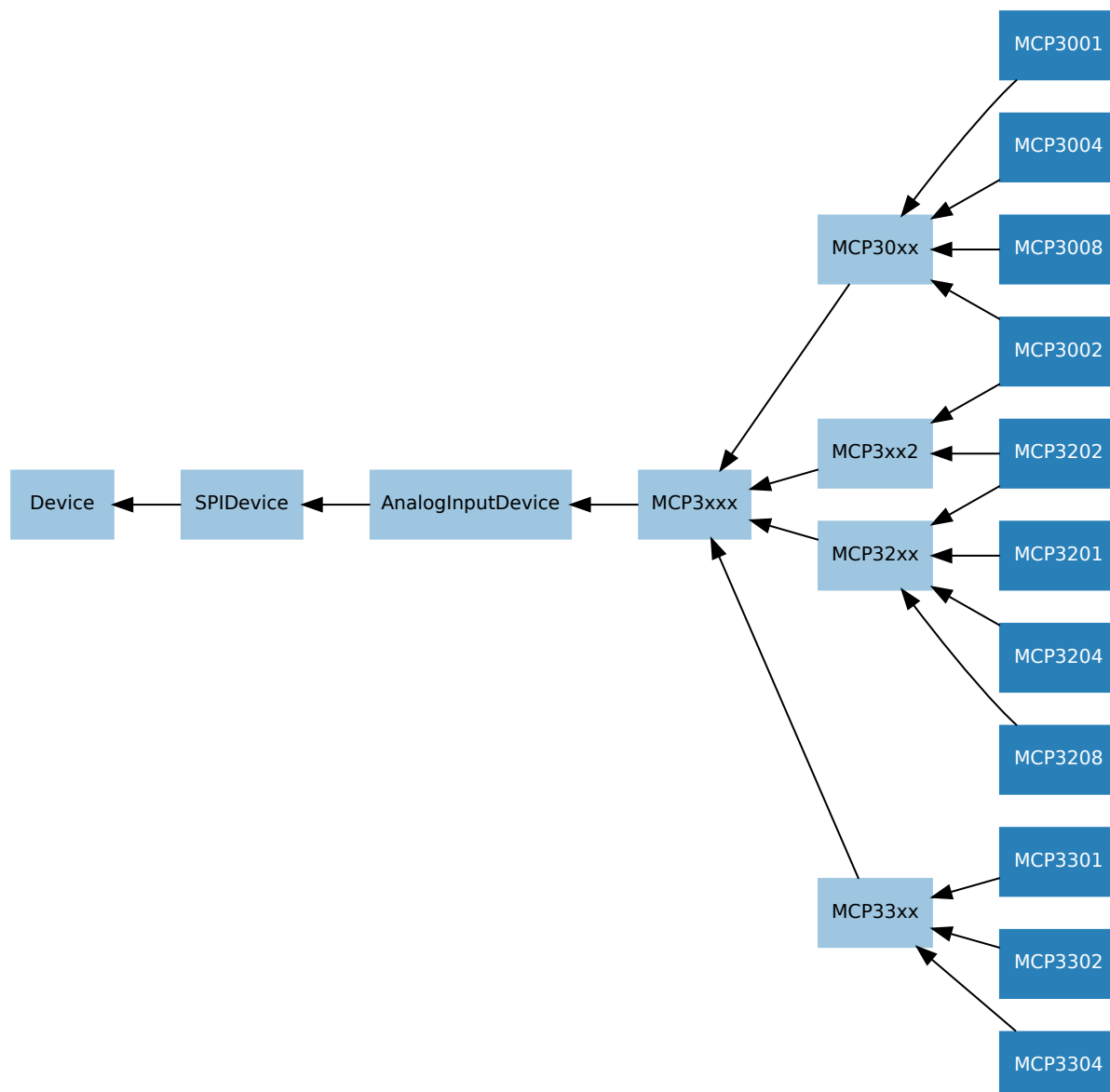
value

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).

14.3 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):

²¹¹ <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.

14.4 AnalogInputDevice

class `gpiozero.AnalogInputDevice` (*bits*, *max_voltage=3.3*, ***spi_args*)

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

Typical analog input devices are [analog to digital converters](#)²¹² (ADCs). Several classes are provided for specific ADC chips, including [MCP3004](#) (page 106), [MCP3008](#) (page 107), [MCP3204](#) (page 107), and [MCP3208](#) (page 108).

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

```
from gpiozero import MCP3008
```

(continues on next page)

²¹² https://en.wikipedia.org/wiki/Analog-to-digital_converter

(continued from previous page)

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

The *value* (page 111) 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 88) like so:

```
from gpiozero import MCP3008, PWMLED

pot = MCP3008(0)
led = PWMLED(17)
led.source = pot.values
```

The *voltage* (page 111) attribute reports values between 0.0 and *max_voltage* (which defaults to 3.3, the logic level of the GPIO pins).

bits

The bit-resolution of the device/channel.

max_voltage

The voltage required to set the device's value to 1.

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).

voltage

The current voltage read from the device. This will be a value between 0 and the *max_voltage* parameter specified in the constructor.

14.5 SPIDevice

class `gpiozero.SPIDevice` (***spi_args*)

Extends *Device* (page 161). Represents a device that communicates via the SPI protocol.

See *SPI keyword args* (page 105) 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()
```

(continues on next page)

(continued from previous page)

```
>>> led = LED(16)
>>> led.blink()
```

Device (page 161) descendants can also be used as context managers using the `with`²¹³ 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 111) method). Once a device is closed you can no longer use any other methods or properties to control or query the device.

²¹³ https://docs.python.org/3.5/reference/compound_stmts.html#with

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 73) and *API - Output Devices* (page 87) 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.

15.1 LEDBoard

class gpiozero.LEDBoard(*pins, pwm=False, active_high=True, initial_value=False,
pin_factory=None, **named_pins)

Extends *LEDCollection* (page 152) 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*²¹⁴) – 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 113) instances to create trees of LEDs.
- **pwm** (*bool*²¹⁵) – If True, construct *PWMLED* (page 88) instances for each pin. If False (the default), construct regular *LED* (page 87) instances. This parameter can only be specified as a keyword parameter.

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

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

- **active_high** (*bool*²¹⁶) – If `True` (the default), the `on()` (page 115) method will set all the associated pins to HIGH. If `False`, the `on()` (page 115) method will set all pins to LOW (the `off()` (page 115) method always does the opposite). This parameter can only be specified as a keyword parameter.
- **initial_value** (*bool*²¹⁷) – 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 180)) – See *API - Pins* (page 177) 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 113) 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*²¹⁸) – Number of seconds on. Defaults to 1 second.
- **off_time** (*float*²¹⁹) – Number of seconds off. Defaults to 1 second.
- **fade_in_time** (*float*²²⁰) – Number of seconds to spend fading in. Defaults to 0. Must be 0 if `pwm` was `False` when the class was constructed (*ValueError*²²¹ will be raised if not).
- **fade_out_time** (*float*²²²) – Number of seconds to spend fading out. Defaults to 0. Must be 0 if `pwm` was `False` when the class was constructed (*ValueError*²²³ will be raised if not).
- **n** (*int*²²⁴) – Number of times to blink; `None` (the default) means forever.
- **background** (*bool*²²⁵) – 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:

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

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

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

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

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

²²¹ <https://docs.python.org/3.5/library/exceptions.html#ValueError>

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

²²³ <https://docs.python.org/3.5/library/exceptions.html#ValueError>

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

²²⁵ <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 161) descendants can also be used as context managers using the `with`²²⁶ 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*²²⁷) – Number of seconds to spend fading in. Defaults to 1.
- **fade_out_time** (*float*²²⁸) – Number of seconds to spend fading out. Defaults to 1.
- **n** (*int*²²⁹) – Number of times to blink; None (the default) means forever.
- **background** (*bool*²³⁰) – 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.

closed

Returns True if the device is closed (see the `close()` (page 114) 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 116). Unlike *value* (page 116), this is *always* a boolean.

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 116).

source_delay

The delay (measured in seconds) in the loop used to read values from *source* (page 115). Defaults to

²²⁶ https://docs.python.org/3.5/reference/compound_stmts.html#with

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

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

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

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

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*.

15.2 LEDBarGraph

class gpiozero.LEDBarGraph(*pins, pwm=False, active_high=True, initial_value=0, pin_factory=None)

Extends [LEDCollection](#) (page 152) 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 117) and [values](#) (page 117) 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](#)²³¹) – Specify the GPIO pins that the LEDs of the bar graph are attached to. You can designate as many pins as necessary.
- **pwm** ([bool](#)²³²) – If `True`, construct [PWMLLED](#) (page 88) instances for each pin. If `False` (the default), construct regular [LED](#) (page 87) instances. This parameter can only be specified as a keyword parameter.
- **active_high** ([bool](#)²³³) – If `True` (the default), the [on\(\)](#) (page 117) method will set all the associated pins to HIGH. If `False`, the [on\(\)](#) (page 117) method will set all pins to LOW (the [off\(\)](#) (page 117) method always does the opposite). This parameter can only be specified as a keyword parameter.
- **initial_value** ([float](#)²³⁴) – The initial [value](#) (page 117) 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.

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

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

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

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

- **pin_factory** ([Factory](#) (page 180)) – See [API - Pins](#) (page 177) 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.

closed

Returns `True` if the device is closed (see the `close()` method). Once a device is closed you can no longer use any other methods or properties to control or query the device.

is_active

Returns `True` if the device is currently active and `False` otherwise. This property is usually derived from `value` (page 117). Unlike `value` (page 117), this is *always* a boolean.

leds

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

lit_count

The number of LEDs on the bar graph actually lit up. Note that just like `value`, this can be negative if the LEDs are lit from last to first.

source

The iterable to use as a source of values for `value` (page 117).

source_delay

The delay (measured in seconds) in the loop used to read values from `source` (page 117). 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 117) is effectively $-1 < \text{value} \leq 1$.

values

An infinite iterator of values read from `value`.

15.3 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 153) and represents a generic button board or collection of buttons.

Parameters

- ***pins** (*int*²³⁵) – Specify the GPIO pins that the buttons of the board are attached to. You can designate as many pins as necessary.
- **pull_up** (*bool*²³⁶) – 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*²³⁷) – 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*²³⁸) – The length of time (in seconds) to wait after any button is pushed, until executing the *when_held* (page 119) handler. Defaults to 1. This parameter can only be specified as a keyword parameter.
- **hold_repeat** (*bool*²³⁹) – If `True`, the *when_held* (page 119) handler will be repeatedly executed as long as any buttons remain held, every *hold_time* seconds. If `False` (the default) the *when_held* (page 119) handler will be only be executed once per hold. This parameter can only be specified as a keyword parameter.
- **pin_factory** (*Factory* (page 180)) – See *API - Pins* (page 177) 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*²⁴⁰) – 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*²⁴¹) – 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*²⁴²) – 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*²⁴³) – 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`.

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

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

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

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

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

²⁴⁰ <https://docs.python.org/3.5/library/functions.html#float>

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

²⁴² <https://docs.python.org/3.5/library/functions.html#float>

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

closed

Returns `True` if the device is closed (see the `close()` method). Once a device is closed you can no longer use any other methods or properties to control or query the device.

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 119) event rather than when the device activated, in contrast to `active_time` (page 163). If the device is not currently held, this is `None`.

hold_repeat

If `True`, `when_held` (page 119) will be executed repeatedly with `hold_time` (page 119) 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 119) handler. If `hold_repeat` (page 119) is `True`, this is also the length of time between invocations of `when_held` (page 119).

inactive_time

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

is_active

Returns `True` if the device is currently active and `False` otherwise. This property is usually derived from `value` (page 119). Unlike `value` (page 119), this is *always* a boolean.

is_held

When `True`, the device has been active for at least `hold_time` (page 119) seconds.

is_pressed

Returns `True` if the device is currently active and `False` otherwise. This property is usually derived from `value` (page 119). Unlike `value` (page 119), this is *always* a boolean.

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.

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.

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 119) 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.

15.4 TrafficLights

class gpiozero.TrafficLights(*red=None, amber=None, green=None, pwm=False, initial_value=False, yellow=None, pin_factory=None*)

Extends *LEDBoard* (page 113) 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*²⁴⁴) – The GPIO pin that the red LED is attached to.
- **amber** (*int*²⁴⁵) – The GPIO pin that the amber LED is attached to.
- **green** (*int*²⁴⁶) – The GPIO pin that the green LED is attached to.
- **pwm** (*bool*²⁴⁷) – If `True`, construct *PWMLED* (page 88) instances to represent each LED. If `False` (the default), construct regular *LED* (page 87) instances.
- **initial_value** (*bool*²⁴⁸) – 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*²⁴⁹) – 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*.

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

²⁴⁵ <https://docs.python.org/3.5/library/functions.html#int>

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

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

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

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

- **pin_factory** ([Factory](#) (page 180)) – See [API - Pins](#) (page 177) 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*²⁵⁰) – Number of seconds on. Defaults to 1 second.
- **off_time** (*float*²⁵¹) – Number of seconds off. Defaults to 1 second.
- **fade_in_time** (*float*²⁵²) – Number of seconds to spend fading in. Defaults to 0. Must be 0 if `pwm` was `False` when the class was constructed (`ValueError`²⁵³ will be raised if not).
- **fade_out_time** (*float*²⁵⁴) – Number of seconds to spend fading out. Defaults to 0. Must be 0 if `pwm` was `False` when the class was constructed (`ValueError`²⁵⁵ will be raised if not).
- **n** (*int*²⁵⁶) – Number of times to blink; `None` (the default) means forever.
- **background** (*bool*²⁵⁷) – 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 161) descendants can also be used as context managers using the `with`²⁵⁸ statement. For example:

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

(continues on next page)

²⁵⁰ <https://docs.python.org/3.5/library/functions.html#float>

²⁵¹ <https://docs.python.org/3.5/library/functions.html#float>

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

²⁵³ <https://docs.python.org/3.5/library/exceptions.html#ValueError>

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

²⁵⁵ <https://docs.python.org/3.5/library/exceptions.html#ValueError>

²⁵⁶ <https://docs.python.org/3.5/library/functions.html#int>

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

²⁵⁸ https://docs.python.org/3.5/reference/compound_stmts.html#with

(continued from previous page)

```
>>> 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*²⁵⁹) – Number of seconds to spend fading in. Defaults to 1.
- **fade_out_time** (*float*²⁶⁰) – Number of seconds to spend fading out. Defaults to 1.
- **n** (*int*²⁶¹) – Number of times to blink; None (the default) means forever.
- **background** (*bool*²⁶²) – 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.

closedReturns True if the device is closed (see the `close()` (page 121) 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 122). Unlike *value* (page 122), this is *always* a boolean.**leds**

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

sourceThe 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.

valuesAn infinite iterator of values read from *value*.

²⁵⁹ <https://docs.python.org/3.5/library/functions.html#float>²⁶⁰ <https://docs.python.org/3.5/library/functions.html#float>²⁶¹ <https://docs.python.org/3.5/library/functions.html#int>²⁶² <https://docs.python.org/3.5/library/functions.html#bool>

15.5 LedBorg

class gpiozero.LedBorg (*initial_value=(0, 0, 0), pwm=True, pin_factory=None*)

Extends *RGBLED* (page 90) for the *PiBorg LedBorg*²⁶³: 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*²⁶⁴) – The initial color for the LedBorg. Defaults to black (0, 0, 0).
- **pwm** (*bool*²⁶⁵) – If True (the default), construct *PWMLED* (page 88) instances for each component of the LedBorg. If False, construct regular *LED* (page 87) instances, which prevents smooth color graduations.
- **pin_factory** (*Factory* (page 180)) – See *API - Pins* (page 177) 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*²⁶⁶) – Number of seconds on. Defaults to 1 second.
- **off_time** (*float*²⁶⁷) – Number of seconds off. Defaults to 1 second.
- **fade_in_time** (*float*²⁶⁸) – Number of seconds to spend fading in. Defaults to 0. Must be 0 if *pwm* was False when the class was constructed (*ValueError*²⁶⁹ will be raised if not).
- **fade_out_time** (*float*²⁷⁰) – Number of seconds to spend fading out. Defaults to 0. Must be 0 if *pwm* was False when the class was constructed (*ValueError*²⁷¹ will be raised if not).
- **on_color** (*tuple*²⁷²) – The color to use when the LED is “on”. Defaults to white.
- **off_color** (*tuple*²⁷³) – The color to use when the LED is “off”. Defaults to black.
- **n** (*int*²⁷⁴) – Number of times to blink; None (the default) means forever.
- **background** (*bool*²⁷⁵) – 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).

²⁶³ <https://www.piborg.org/ledborg>

²⁶⁴ <https://docs.python.org/3.5/library/stdtypes.html#tuple>

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

²⁶⁶ <https://docs.python.org/3.5/library/functions.html#float>

²⁶⁷ <https://docs.python.org/3.5/library/functions.html#float>

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

²⁶⁹ <https://docs.python.org/3.5/library/exceptions.html#ValueError>

²⁷⁰ <https://docs.python.org/3.5/library/functions.html#float>

²⁷¹ <https://docs.python.org/3.5/library/exceptions.html#ValueError>

²⁷² <https://docs.python.org/3.5/library/stdtypes.html#tuple>

²⁷³ <https://docs.python.org/3.5/library/stdtypes.html#tuple>

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

²⁷⁵ <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 161) descendents can also be used as context managers using the `with`²⁷⁶ statement. For example:

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

off()

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

on()

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

pulse (*fade_in_time=1, fade_out_time=1, on_color=(1, 1, 1), off_color=(0, 0, 0), n=None, background=True*)

Make the device fade in and out repeatedly.

Parameters

- **fade_in_time** (*float*²⁷⁷) – Number of seconds to spend fading in. Defaults to 1.
- **fade_out_time** (*float*²⁷⁸) – Number of seconds to spend fading out. Defaults to 1.
- **on_color** (*tuple*²⁷⁹) – The color to use when the LED is “on”. Defaults to white.
- **off_color** (*tuple*²⁸⁰) – The color to use when the LED is “off”. Defaults to black.
- **n** (*int*²⁸¹) – Number of times to pulse; None (the default) means forever.
- **background** (*bool*²⁸²) – 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).

²⁷⁶ https://docs.python.org/3.5/reference/compound_stmts.html#with

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

²⁷⁸ <https://docs.python.org/3.5/library/functions.html#float>

²⁷⁹ <https://docs.python.org/3.5/library/stdtypes.html#tuple>

²⁸⁰ <https://docs.python.org/3.5/library/stdtypes.html#tuple>

²⁸¹ <https://docs.python.org/3.5/library/functions.html#int>

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

toggle()

Toggle the state of the device. If the device is currently off (*value* (page 125) is (0, 0, 0)), this changes it to “fully” on (*value* (page 125) is (1, 1, 1)). If the device has a specific color, this method inverts the color.

closed

Returns `True` if the device is closed (see the `close()` (page 123) method). Once a device is closed you can no longer use any other methods or properties to control or query the device.

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 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

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*.

15.6 PiLITER

class `gpiozero.PiLiter` (*pwm=False, initial_value=False, pin_factory=None*)

Extends *LEDBoard* (page 113) for the *Ciseco Pi-LITEr*²⁸³: a strip of 8 very bright LEDs.

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

```
from gpiozero import PiLiter

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

Parameters

- **pwm** (*bool*²⁸⁴) – If `True`, construct *PWMLED* (page 88) instances for each pin. If `False` (the default), construct regular *LED* (page 87) instances.

²⁸³ <http://shop.ciseco.co.uk/pi-liter-8-led-strip-for-the-raspberry-pi/>

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

- **initial_value** (*bool*²⁸⁵) – 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 180)) – See *API - Pins* (page 177) 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*²⁸⁶) – Number of seconds on. Defaults to 1 second.
- **off_time** (*float*²⁸⁷) – Number of seconds off. Defaults to 1 second.
- **fade_in_time** (*float*²⁸⁸) – Number of seconds to spend fading in. Defaults to 0. Must be 0 if `pwm` was `False` when the class was constructed (*ValueError*²⁸⁹ will be raised if not).
- **fade_out_time** (*float*²⁹⁰) – Number of seconds to spend fading out. Defaults to 0. Must be 0 if `pwm` was `False` when the class was constructed (*ValueError*²⁹¹ will be raised if not).
- **n** (*int*²⁹²) – Number of times to blink; `None` (the default) means forever.
- **background** (*bool*²⁹³) – 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 161) descendants can also be used as context managers using the `with`²⁹⁴ statement. For example:

²⁸⁵ <https://docs.python.org/3.5/library/functions.html#bool>

²⁸⁶ <https://docs.python.org/3.5/library/functions.html#float>

²⁸⁷ <https://docs.python.org/3.5/library/functions.html#float>

²⁸⁸ <https://docs.python.org/3.5/library/functions.html#float>

²⁸⁹ <https://docs.python.org/3.5/library/exceptions.html#ValueError>

²⁹⁰ <https://docs.python.org/3.5/library/functions.html#float>

²⁹¹ <https://docs.python.org/3.5/library/exceptions.html#ValueError>

²⁹² <https://docs.python.org/3.5/library/functions.html#int>

²⁹³ <https://docs.python.org/3.5/library/functions.html#bool>

²⁹⁴ https://docs.python.org/3.5/reference/compound_stmts.html#with

```
>>> 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*²⁹⁵) – Number of seconds to spend fading in. Defaults to 1.
- **fade_out_time** (*float*²⁹⁶) – Number of seconds to spend fading out. Defaults to 1.
- **n** (*int*²⁹⁷) – Number of times to blink; None (the default) means forever.
- **background** (*bool*²⁹⁸) – 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.

closed

Returns True if the device is closed (see the `close()` (page 126) 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 127). Unlike *value* (page 127), this is *always* a boolean.

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 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.

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*.

²⁹⁵ <https://docs.python.org/3.5/library/functions.html#float>

²⁹⁶ <https://docs.python.org/3.5/library/functions.html#float>

²⁹⁷ <https://docs.python.org/3.5/library/functions.html#int>

²⁹⁸ <https://docs.python.org/3.5/library/functions.html#bool>

15.7 PiLITEr Bar Graph

class gpiozero.PiLiterBarGraph (*pwm=False, initial_value=0.0, pin_factory=None*)

Extends [LEDBarGraph](#) (page 116) to treat the [Ciseco Pi-LITEr](#)²⁹⁹ 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*³⁰⁰) – If `True`, construct [PWMLed](#) (page 88) instances for each pin. If `False` (the default), construct regular [LED](#) (page 87) instances.
- **initial_value** (*float*³⁰¹) – The initial *value* (page 128) of the graph given as a float between -1 and +1. Defaults to 0.0.
- **pin_factory** ([Factory](#) (page 180)) – See [API - Pins](#) (page 177) 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.

closed

Returns `True` if the device is closed (see the `close()` method). Once a device is closed you can no longer use any other methods or properties to control or query the device.

is_active

Returns `True` if the device is currently active and `False` otherwise. This property is usually derived from *value* (page 128). Unlike *value* (page 128), this is *always* a boolean.

leds

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

lit_count

The number of LEDs on the bar graph actually lit up. Note that just like *value*, this can be negative if the LEDs are lit from last to first.

source

The iterable to use as a source of values for *value* (page 128).

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

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.

²⁹⁹ <http://shop.ciseco.co.uk/pi-liter-8-led-strip-for-the-raspberry-pi/>

³⁰⁰ <https://docs.python.org/3.5/library/functions.html#bool>

³⁰¹ <https://docs.python.org/3.5/library/functions.html#float>

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 128) is effectively $-1 < \text{value} \leq 1$.

values

An infinite iterator of values read from *value*.

15.8 PI-TRAFFIC

class gpiozero.**PiTraffic** (*pwm=False, initial_value=False, pin_factory=None*)

Extends *TrafficLights* (page 120) for the Low Voltage Labs PI-TRAFFIC³⁰² 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 120).

Parameters

- **pwm** (*bool*³⁰³) – If *True*, construct *PWMLED* (page 88) instances to represent each LED. If *False* (the default), construct regular *LED* (page 87) instances.
- **initial_value** (*bool*³⁰⁴) – 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 180)) – See *API - Pins* (page 177) 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*³⁰⁵) – Number of seconds on. Defaults to 1 second.
- **off_time** (*float*³⁰⁶) – Number of seconds off. Defaults to 1 second.

³⁰² <http://lowvoltage labs.com/products/pi-traffic/>

³⁰³ <https://docs.python.org/3.5/library/functions.html#bool>

³⁰⁴ <https://docs.python.org/3.5/library/functions.html#bool>

³⁰⁵ <https://docs.python.org/3.5/library/functions.html#float>

³⁰⁶ <https://docs.python.org/3.5/library/functions.html#float>

- **fade_in_time** (*float*³⁰⁷) – Number of seconds to spend fading in. Defaults to 0. Must be 0 if `pwm` was `False` when the class was constructed (`ValueError`³⁰⁸ will be raised if not).
- **fade_out_time** (*float*³⁰⁹) – Number of seconds to spend fading out. Defaults to 0. Must be 0 if `pwm` was `False` when the class was constructed (`ValueError`³¹⁰ will be raised if not).
- **n** (*int*³¹¹) – Number of times to blink; `None` (the default) means forever.
- **background** (*bool*³¹²) – 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 161) descendants can also be used as context managers using the `with`³¹³ 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

³⁰⁷ <https://docs.python.org/3.5/library/functions.html#float>

³⁰⁸ <https://docs.python.org/3.5/library/exceptions.html#ValueError>

³⁰⁹ <https://docs.python.org/3.5/library/functions.html#float>

³¹⁰ <https://docs.python.org/3.5/library/exceptions.html#ValueError>

³¹¹ <https://docs.python.org/3.5/library/functions.html#int>

³¹² <https://docs.python.org/3.5/library/functions.html#bool>

³¹³ https://docs.python.org/3.5/reference/compound_stmts.html#with

- **fade_in_time** (*float*³¹⁴) – Number of seconds to spend fading in. Defaults to 1.
- **fade_out_time** (*float*³¹⁵) – Number of seconds to spend fading out. Defaults to 1.
- **n** (*int*³¹⁶) – Number of times to blink; `None` (the default) means forever.
- **background** (*bool*³¹⁷) – 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.

closed

Returns `True` if the device is closed (see the `close()` (page 130) 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 131). Unlike *value* (page 131), this is *always* a boolean.

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 131).

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.

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*.

15.9 Pi-Stop

class `gpiozero.PiStop` (*location=None, pwm=False, initial_value=False, pin_factory=None*)

Extends *TrafficLights* (page 120) for the *PiHardware Pi-Stop*³¹⁸: 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*³¹⁹) – The *location*³²⁰ on the GPIO header to which the Pi-Stop is connected. Must be one of: A, A+, B, B+, C, D.

³¹⁴ <https://docs.python.org/3.5/library/functions.html#float>

³¹⁵ <https://docs.python.org/3.5/library/functions.html#float>

³¹⁶ <https://docs.python.org/3.5/library/functions.html#int>

³¹⁷ <https://docs.python.org/3.5/library/functions.html#bool>

³¹⁸ <https://pihw.wordpress.com/meltwaters-pi-hardware-kits/pi-stop/>

³¹⁹ <https://docs.python.org/3.5/library/stdtypes.html#str>

³²⁰ https://github.com/PiHw/Pi-Stop/blob/master/markdown_source/markdown/Discover-PiStop.md

- **pwm** (*bool*³²¹) – If `True`, construct *PWMLED* (page 88) instances to represent each LED. If `False` (the default), construct regular *LED* (page 87) instances.
- **initial_value** (*bool*³²²) – 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 180)) – See *API - Pins* (page 177) 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*³²³) – Number of seconds on. Defaults to 1 second.
- **off_time** (*float*³²⁴) – Number of seconds off. Defaults to 1 second.
- **fade_in_time** (*float*³²⁵) – Number of seconds to spend fading in. Defaults to 0. Must be 0 if `pwm` was `False` when the class was constructed (*ValueError*³²⁶ will be raised if not).
- **fade_out_time** (*float*³²⁷) – Number of seconds to spend fading out. Defaults to 0. Must be 0 if `pwm` was `False` when the class was constructed (*ValueError*³²⁸ will be raised if not).
- **n** (*int*³²⁹) – Number of times to blink; `None` (the default) means forever.
- **background** (*bool*³³⁰) – 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()
```

³²¹ <https://docs.python.org/3.5/library/functions.html#bool>

³²² <https://docs.python.org/3.5/library/functions.html#bool>

³²³ <https://docs.python.org/3.5/library/functions.html#float>

³²⁴ <https://docs.python.org/3.5/library/functions.html#float>

³²⁵ <https://docs.python.org/3.5/library/functions.html#float>

³²⁶ <https://docs.python.org/3.5/library/exceptions.html#ValueError>

³²⁷ <https://docs.python.org/3.5/library/functions.html#float>

³²⁸ <https://docs.python.org/3.5/library/exceptions.html#ValueError>

³²⁹ <https://docs.python.org/3.5/library/functions.html#int>

³³⁰ <https://docs.python.org/3.5/library/functions.html#bool>

Device (page 161) descendants can also be used as context managers using the `with`³³¹ 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*³³²) – Number of seconds to spend fading in. Defaults to 1.
- **fade_out_time** (*float*³³³) – Number of seconds to spend fading out. Defaults to 1.
- **n** (*int*³³⁴) – Number of times to blink; `None` (the default) means forever.
- **background** (*bool*³³⁵) – 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.

closed

Returns `True` if the device is closed (see the `close()` (page 132) 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 133). Unlike *value* (page 133), this is *always* a boolean.

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*.

³³¹ https://docs.python.org/3.5/reference/compound_stmts.html#with

³³² <https://docs.python.org/3.5/library/functions.html#float>

³³³ <https://docs.python.org/3.5/library/functions.html#float>

³³⁴ <https://docs.python.org/3.5/library/functions.html#int>

³³⁵ <https://docs.python.org/3.5/library/functions.html#bool>

15.10 TrafficLightsBuzzer

class gpiozero.**TrafficLightsBuzzer** (*lights, buzzer, button, pin_factory=None*)

Extends *CompositeOutputDevice* (page 152) and is a generic class for HATs with traffic lights, a button and a buzzer.

Parameters

- **lights** (*TrafficLights* (page 120)) – An instance of *TrafficLights* (page 120) representing the traffic lights of the HAT.
- **buzzer** (*Buzzer* (page 92)) – An instance of *Buzzer* (page 92) representing the buzzer on the HAT.
- **button** (*Button* (page 73)) – An instance of *Button* (page 73) representing the button on the HAT.
- **pin_factory** (*Factory* (page 180)) – See *API - Pins* (page 177) 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.

closed

Returns *True* if the device is closed (see the *close* () method). Once a device is closed you can no longer use any other methods or properties to control or query the device.

is_active

Returns *True* if the device is currently active and *False* otherwise. This property is usually derived from *value* (page 134). Unlike *value* (page 134), this is *always* a boolean.

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*.

15.11 Fish Dish

class gpiozero.**FishDish** (*pwm=False, pin_factory=None*)

Extends *TrafficLightsBuzzer* (page 134) for the *Pi Supply FishDish*³³⁶: 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:

³³⁶ <https://www.pi-supply.com/product/fish-dish-raspberry-pi-led-buzzer-board/>

```
from gpiozero import FishDish

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

Parameters

- **pwm** (*bool*³³⁷) – If `True`, construct *PWMLED* (page 88) instances to represent each LED. If `False` (the default), construct regular *LED* (page 87) instances.
- **pin_factory** (*Factory* (page 180)) – See *API - Pins* (page 177) 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.

closed

Returns `True` if the device is closed (see the `close()` method). Once a device is closed you can no longer use any other methods or properties to control or query the device.

is_active

Returns `True` if the device is currently active and `False` otherwise. This property is usually derived from *value* (page 135). Unlike *value* (page 135), this is *always* a boolean.

source

The iterable to use as a source of values for *value* (page 135).

source_delay

The delay (measured in seconds) in the loop used to read values from *source* (page 135). 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*.

15.12 Traffic HAT

class `gpiozero.TrafficHat` (*pwm=False, pin_factory=None*)

Extends *TrafficLightsBuzzer* (page 134) for the *Ryanteck Traffic HAT*³³⁸: 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()
```

(continues on next page)

³³⁷ <https://docs.python.org/3.5/library/functions.html#bool>

³³⁸ <https://ryanteck.uk/hats/1-traffic-hat-0635648607122.html>

(continued from previous page)

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

Parameters

- **pwm** (*bool*³³⁹) – If `True`, construct *PWMLED* (page 88) instances to represent each LED. If `False` (the default), construct regular *LED* (page 87) instances.
- **pin_factory** (*Factory* (page 180)) – See *API - Pins* (page 177) 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.

closed

Returns `True` if the device is closed (see the `close()` method). Once a device is closed you can no longer use any other methods or properties to control or query the device.

is_active

Returns `True` if the device is currently active and `False` otherwise. This property is usually derived from *value* (page 136). Unlike *value* (page 136), this is *always* a boolean.

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*.

15.13 Robot

class `gpiozero.Robot` (*left=None, right=None, pin_factory=None*)

Extends *CompositeDevice* (page 153) to represent a generic dual-motor robot.

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

```
from gpiozero import Robot

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

Parameters

³³⁹ <https://docs.python.org/3.5/library/functions.html#bool>

- **left** (*tuple*³⁴⁰) – A tuple of two GPIO pins representing the forward and backward inputs of the left motor’s controller.
- **right** (*tuple*³⁴¹) – A tuple of two GPIO pins representing the forward and backward inputs of the right motor’s controller.
- **pin_factory** (*Factory* (page 180)) – See *API - Pins* (page 177) for more information (this is an advanced feature which most users can ignore).

backward (*speed=1, **kwargs*)

Drive the robot backward by running both motors backward.

Parameters

- **speed** (*float*³⁴²) – Speed at which to drive the motors, as a value between 0 (stopped) and 1 (full speed). The default is 1.
- **curve_left** (*float*³⁴³) – The amount to curve left while moving backwards, by driving the left motor at a slower speed. Maximum *curve_left* is 1, the default is 0 (no curve). This parameter can only be specified as a keyword parameter, and is mutually exclusive with *curve_right*.
- **curve_right** (*float*³⁴⁴) – The amount to curve right while moving backwards, by driving the right motor at a slower speed. Maximum *curve_right* is 1, the default is 0 (no curve). This parameter can only be specified as a keyword parameter, and is mutually exclusive with *curve_left*.

forward (*speed=1, **kwargs*)

Drive the robot forward by running both motors forward.

Parameters

- **speed** (*float*³⁴⁵) – Speed at which to drive the motors, as a value between 0 (stopped) and 1 (full speed). The default is 1.
- **curve_left** (*float*³⁴⁶) – The amount to curve left while moving forwards, by driving the left motor at a slower speed. Maximum *curve_left* is 1, the default is 0 (no curve). This parameter can only be specified as a keyword parameter, and is mutually exclusive with *curve_right*.
- **curve_right** (*float*³⁴⁷) – The amount to curve right while moving forwards, by driving the right motor at a slower speed. Maximum *curve_right* is 1, the default is 0 (no curve). This parameter can only be specified as a keyword parameter, and is mutually exclusive with *curve_left*.

left (*speed=1*)

Make the robot turn left by running the right motor forward and left motor backward.

Parameters **speed** (*float*³⁴⁸) – 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.

³⁴⁰ <https://docs.python.org/3.5/library/stdtypes.html#tuple>

³⁴¹ <https://docs.python.org/3.5/library/stdtypes.html#tuple>

³⁴² <https://docs.python.org/3.5/library/functions.html#float>

³⁴³ <https://docs.python.org/3.5/library/functions.html#float>

³⁴⁴ <https://docs.python.org/3.5/library/functions.html#float>

³⁴⁵ <https://docs.python.org/3.5/library/functions.html#float>

³⁴⁶ <https://docs.python.org/3.5/library/functions.html#float>

³⁴⁷ <https://docs.python.org/3.5/library/functions.html#float>

³⁴⁸ <https://docs.python.org/3.5/library/functions.html#float>

Parameters `speed` (*float*³⁴⁹) – 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.

closed

Returns `True` if the device is closed (see the `close()` method). Once a device is closed you can no longer use any other methods or properties to control or query the device.

is_active

Returns `True` if the device is currently active and `False` otherwise. This property is usually derived from `value` (page 138). Unlike `value` (page 138), this is *always* a boolean.

source

The iterable to use as a source of values for `value` (page 138).

source_delay

The delay (measured in seconds) in the loop used to read values from `source` (page 138). 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`.

15.14 PhaseEnableRobot

class `gpiozero.PhaseEnableRobot` (*left=None, right=None, pin_factory=None*)

Extends `CompositeDevice` (page 153) to represent a dual-motor robot based around a Phase/Enable motor board.

This class is constructed with two tuples representing the phase (direction) and enable (speed) pins of the left and right controllers respectively. For example, if the left motor's controller is connected to GPIOs 12 and 5, while the right motor's controller is connected to GPIOs 13 and 6 so the following example will drive the robot forward:

```
from gpiozero import PhaseEnableRobot

robot = PhaseEnableRobot(left=(5, 12), right=(6, 13))
robot.forward()
```

Parameters

- **left** (*tuple*³⁵⁰) – A tuple of two GPIO pins representing the phase and enable inputs of the left motor's controller.
- **right** (*tuple*³⁵¹) – A tuple of two GPIO pins representing the phase and enable inputs of the right motor's controller.
- **pin_factory** (`Factory` (page 180)) – See *API - Pins* (page 177) 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.

³⁴⁹ <https://docs.python.org/3.5/library/functions.html#float>

³⁵⁰ <https://docs.python.org/3.5/library/stdtypes.html#tuple>

³⁵¹ <https://docs.python.org/3.5/library/stdtypes.html#tuple>

Parameters `speed` ([*float*](#)³⁵²) – 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*](#)³⁵³) – 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*](#)³⁵⁴) – 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*](#)³⁵⁵) – 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.

closed

Returns `True` if the device is closed (see the `close()` method). Once a device is closed you can no longer use any other methods or properties to control or query the device.

is_active

Returns `True` if the device is currently active and `False` otherwise. This property is usually derived from `value` (page 139). Unlike `value` (page 139), this is *always* a boolean.

source

The iterable to use as a source of values for `value` (page 139).

source_delay

The delay (measured in seconds) in the loop used to read values from `source` (page 139). Defaults to 0.01 seconds which is generally sufficient to keep CPU usage to a minimum while providing adequate responsiveness.

value

Returns a tuple of two floating point values (-1 to 1) representing the speeds of the robot’s two motors (left and right). This property can also be set to alter the speed of both motors.

values

An infinite iterator of values read from `value`.

15.15 Ryantek MCB Robot

class `gpiozero.RyantekRobot` (`pin_factory=None`)

Extends `Robot` (page 136) for the [Ryantek motor controller board](#)³⁵⁶.

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

³⁵² <https://docs.python.org/3.5/library/functions.html#float>

³⁵³ <https://docs.python.org/3.5/library/functions.html#float>

³⁵⁴ <https://docs.python.org/3.5/library/functions.html#float>

³⁵⁵ <https://docs.python.org/3.5/library/functions.html#float>

³⁵⁶ <https://ryantek.uk/add-ons/6-ryantek-rpi-motor-controller-board-0635648607160.html>

```
from gpiozero import RyanteckRobot

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

Parameters **pin_factory** ([Factory](#) (page 180)) – See [API - Pins](#) (page 177) for more information (this is an advanced feature which most users can ignore).

backward (*speed=1, **kwargs*)

Drive the robot backward by running both motors backward.

Parameters

- **speed** (*float*³⁵⁷) – Speed at which to drive the motors, as a value between 0 (stopped) and 1 (full speed). The default is 1.
- **curve_left** (*float*³⁵⁸) – The amount to curve left while moving backwards, by driving the left motor at a slower speed. Maximum `curve_left` is 1, the default is 0 (no curve). This parameter can only be specified as a keyword parameter, and is mutually exclusive with `curve_right`.
- **curve_right** (*float*³⁵⁹) – The amount to curve right while moving backwards, by driving the right motor at a slower speed. Maximum `curve_right` is 1, the default is 0 (no curve). This parameter can only be specified as a keyword parameter, and is mutually exclusive with `curve_left`.

forward (*speed=1, **kwargs*)

Drive the robot forward by running both motors forward.

Parameters

- **speed** (*float*³⁶⁰) – Speed at which to drive the motors, as a value between 0 (stopped) and 1 (full speed). The default is 1.
- **curve_left** (*float*³⁶¹) – The amount to curve left while moving forwards, by driving the left motor at a slower speed. Maximum `curve_left` is 1, the default is 0 (no curve). This parameter can only be specified as a keyword parameter, and is mutually exclusive with `curve_right`.
- **curve_right** (*float*³⁶²) – The amount to curve right while moving forwards, by driving the right motor at a slower speed. Maximum `curve_right` is 1, the default is 0 (no curve). This parameter can only be specified as a keyword parameter, and is mutually exclusive with `curve_left`.

left (*speed=1*)

Make the robot turn left by running the right motor forward and left motor backward.

Parameters **speed** (*float*³⁶³) – 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.

³⁵⁷ <https://docs.python.org/3.5/library/functions.html#float>

³⁵⁸ <https://docs.python.org/3.5/library/functions.html#float>

³⁵⁹ <https://docs.python.org/3.5/library/functions.html#float>

³⁶⁰ <https://docs.python.org/3.5/library/functions.html#float>

³⁶¹ <https://docs.python.org/3.5/library/functions.html#float>

³⁶² <https://docs.python.org/3.5/library/functions.html#float>

³⁶³ <https://docs.python.org/3.5/library/functions.html#float>

Parameters **speed** (*float*³⁶⁴) – 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.

closed

Returns `True` if the device is closed (see the `close()` method). Once a device is closed you can no longer use any other methods or properties to control or query the device.

is_active

Returns `True` if the device is currently active and `False` otherwise. This property is usually derived from *value* (page 141). Unlike *value* (page 141), this is *always* a boolean.

source

The iterable to use as a source of values for *value* (page 141).

source_delay

The delay (measured in seconds) in the loop used to read values from *source* (page 141). 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*.

15.16 CamJam #3 Kit Robot

class `gpiozero.CamJamKitRobot` (*pin_factory=None*)

Extends *Robot* (page 136) for the *CamJam #3 EduKit*³⁶⁵ 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 180)) – See *API - Pins* (page 177) for more information (this is an advanced feature which most users can ignore).

backward (*speed=1, **kwargs*)

Drive the robot backward by running both motors backward.

Parameters

- **speed** (*float*³⁶⁶) – Speed at which to drive the motors, as a value between 0 (stopped) and 1 (full speed). The default is 1.
- **curve_left** (*float*³⁶⁷) – The amount to curve left while moving backwards, by driving the left motor at a slower speed. Maximum *curve_left* is 1, the default is 0 (no curve). This parameter can only be specified as a keyword parameter, and is mutually exclusive with *curve_right*.

³⁶⁴ <https://docs.python.org/3.5/library/functions.html#float>

³⁶⁵ http://camjam.me/?page_id=1035

³⁶⁶ <https://docs.python.org/3.5/library/functions.html#float>

³⁶⁷ <https://docs.python.org/3.5/library/functions.html#float>

- **curve_right** (*float*³⁶⁸) – The amount to curve right while moving backwards, by driving the right motor at a slower speed. Maximum `curve_right` is 1, the default is 0 (no curve). This parameter can only be specified as a keyword parameter, and is mutually exclusive with `curve_left`.

forward (*speed=1, **kwargs*)

Drive the robot forward by running both motors forward.

Parameters

- **speed** (*float*³⁶⁹) – Speed at which to drive the motors, as a value between 0 (stopped) and 1 (full speed). The default is 1.
- **curve_left** (*float*³⁷⁰) – The amount to curve left while moving forwards, by driving the left motor at a slower speed. Maximum `curve_left` is 1, the default is 0 (no curve). This parameter can only be specified as a keyword parameter, and is mutually exclusive with `curve_right`.
- **curve_right** (*float*³⁷¹) – The amount to curve right while moving forwards, by driving the right motor at a slower speed. Maximum `curve_right` is 1, the default is 0 (no curve). This parameter can only be specified as a keyword parameter, and is mutually exclusive with `curve_left`.

left (*speed=1*)

Make the robot turn left by running the right motor forward and left motor backward.

Parameters **speed** (*float*³⁷²) – 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*³⁷³) – 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.

closed

Returns `True` if the device is closed (see the `close()` method). Once a device is closed you can no longer use any other methods or properties to control or query the device.

is_active

Returns `True` if the device is currently active and `False` otherwise. This property is usually derived from *value* (page 142). Unlike *value* (page 142), this is *always* a boolean.

source

The iterable to use as a source of values for *value* (page 142).

source_delay

The delay (measured in seconds) in the loop used to read values from *source* (page 142). Defaults to 0.01 seconds which is generally sufficient to keep CPU usage to a minimum while providing adequate responsiveness.

³⁶⁸ <https://docs.python.org/3.5/library/functions.html#float>

³⁶⁹ <https://docs.python.org/3.5/library/functions.html#float>

³⁷⁰ <https://docs.python.org/3.5/library/functions.html#float>

³⁷¹ <https://docs.python.org/3.5/library/functions.html#float>

³⁷² <https://docs.python.org/3.5/library/functions.html#float>

³⁷³ <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*.

15.17 Pololu DRV8835 Robot

class gpiozero.PololuDRV8835Robot (*pin_factory=None*)

Extends *PhaseEnableRobot* (page 138) for the Pololu DRV8835 Dual Motor Driver Kit³⁷⁴.

The Pololu DRV8835 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 PololuDRV8835Robot

robot = PololuDRV8835Robot()
robot.forward()
```

Parameters *pin_factory* (*Factory* (page 180)) – See *API - Pins* (page 177) 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*³⁷⁵) – 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*³⁷⁶) – 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*³⁷⁷) – 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*³⁷⁸) – 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.

³⁷⁴ <https://www.pololu.com/product/2753>

³⁷⁵ <https://docs.python.org/3.5/library/functions.html#float>

³⁷⁶ <https://docs.python.org/3.5/library/functions.html#float>

³⁷⁷ <https://docs.python.org/3.5/library/functions.html#float>

³⁷⁸ <https://docs.python.org/3.5/library/functions.html#float>

closed

Returns `True` if the device is closed (see the `close()` method). Once a device is closed you can no longer use any other methods or properties to control or query the device.

is_active

Returns `True` if the device is currently active and `False` otherwise. This property is usually derived from `value` (page 144). Unlike `value` (page 144), this is *always* a boolean.

source

The iterable to use as a source of values for `value` (page 144).

source_delay

The delay (measured in seconds) in the loop used to read values from `source` (page 144). Defaults to 0.01 seconds which is generally sufficient to keep CPU usage to a minimum while providing adequate responsiveness.

value

Returns a tuple of two floating point values (-1 to 1) representing the speeds of the robot's two motors (left and right). This property can also be set to alter the speed of both motors.

values

An infinite iterator of values read from `value`.

15.18 Energenie

class `gpiozero.Energenie` (*socket=None, initial_value=False, pin_factory=None*)

Extends `Device` (page 161) to represent an `Energenie socket`³⁷⁹ 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*³⁸⁰) – Which socket this instance should control. This is an integer number between 1 and 4.
- **initial_value** (*bool*³⁸¹) – 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 180)) – See *API - Pins* (page 177) 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

³⁷⁹ <https://energenie4u.co.uk/index.php/catalogue/product/ENER002-2PI>

³⁸⁰ <https://docs.python.org/3.5/library/functions.html#int>

³⁸¹ <https://docs.python.org/3.5/library/functions.html#bool>

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 161) descendants can also be used as context managers using the `with`³⁸² 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 144) 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 145). Unlike *value* (page 145), this is *always* a boolean.

source

The iterable to use as a source of values for *value* (page 145).

source_delay

The delay (measured in seconds) in the loop used to read values from *source* (page 145). Defaults to 0.01 seconds which is generally sufficient to keep CPU usage to a minimum while providing adequate responsiveness.

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.

values

An infinite iterator of values read from *value*.

15.19 StatusZero

class `gpiozero.StatusZero(*labels, pwm=False, active_high=True, initial_value=False, pin_factory=None)`

Extends *LEDBoard* (page 113) for The Pi Hut’s *STATUS Zero*³⁸³: 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:

³⁸² https://docs.python.org/3.5/reference/compound_stmts.html#with

³⁸³ <https://thepihut.com/statuszero>

```
from gpiozero import StatusZero

status = StatusZero('wifi', 'raining')
status.wifi.green.on()
status.raining.red.on()
```

Parameters

- ***labels** (*str*³⁸⁴) – 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.
- **pin_factory** (*Factory* (page 180)) – See *API - Pins* (page 177) 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*³⁸⁵) – Number of seconds on. Defaults to 1 second.
- **off_time** (*float*³⁸⁶) – Number of seconds off. Defaults to 1 second.
- **fade_in_time** (*float*³⁸⁷) – Number of seconds to spend fading in. Defaults to 0. Must be 0 if `pwm` was `False` when the class was constructed (`ValueError`³⁸⁸ will be raised if not).
- **fade_out_time** (*float*³⁸⁹) – Number of seconds to spend fading out. Defaults to 0. Must be 0 if `pwm` was `False` when the class was constructed (`ValueError`³⁹⁰ will be raised if not).
- **n** (*int*³⁹¹) – Number of times to blink; `None` (the default) means forever.
- **background** (*bool*³⁹²) – 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:

³⁸⁴ <https://docs.python.org/3.5/library/stdtypes.html#str>
³⁸⁵ <https://docs.python.org/3.5/library/functions.html#float>
³⁸⁶ <https://docs.python.org/3.5/library/functions.html#float>
³⁸⁷ <https://docs.python.org/3.5/library/functions.html#float>
³⁸⁸ <https://docs.python.org/3.5/library/exceptions.html#ValueError>
³⁸⁹ <https://docs.python.org/3.5/library/functions.html#float>
³⁹⁰ <https://docs.python.org/3.5/library/exceptions.html#ValueError>
³⁹¹ <https://docs.python.org/3.5/library/functions.html#int>
³⁹² <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 161) descendants can also be used as context managers using the `with`³⁹³ 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*³⁹⁴) – Number of seconds to spend fading in. Defaults to 1.
- **fade_out_time** (*float*³⁹⁵) – Number of seconds to spend fading out. Defaults to 1.
- **n** (*int*³⁹⁶) – Number of times to blink; None (the default) means forever.
- **background** (*bool*³⁹⁷) – 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.

closed

Returns True if the device is closed (see the `close()` (page 146) 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 148). Unlike *value* (page 148), this is *always* a boolean.

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 148).

source_delay

The delay (measured in seconds) in the loop used to read values from *source* (page 147). Defaults to

³⁹³ https://docs.python.org/3.5/reference/compound_stmts.html#with

³⁹⁴ <https://docs.python.org/3.5/library/functions.html#float>

³⁹⁵ <https://docs.python.org/3.5/library/functions.html#float>

³⁹⁶ <https://docs.python.org/3.5/library/functions.html#int>

³⁹⁷ <https://docs.python.org/3.5/library/functions.html#bool>

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*.

15.20 StatusBoard

class gpiozero.**StatusBoard**(*labels, pwm=False, active_high=True, initial_value=False, pin_factory=None)

Extends *CompositeOutputDevice* (page 152) for The Pi Hut’s [STATUS](https://thehiphut.com/status)³⁹⁸ 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:

```
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*³⁹⁹) – 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 180)) – See *API - Pins* (page 177) 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.

closed

Returns *True* if the device is closed (see the *close()* method). Once a device is closed you can no longer use any other methods or properties to control or query the device.

is_active

Returns *True* if the device is currently active and *False* otherwise. This property is usually derived from *value* (page 148). Unlike *value* (page 148), this is *always* a boolean.

source

The iterable to use as a source of values for *value* (page 148).

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.

³⁹⁸ <https://thehiphut.com/status>

³⁹⁹ <https://docs.python.org/3.5/library/stdtypes.html#str>

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*.

15.21 SnowPi

class gpiozero.SnowPi (*pwm=False, initial_value=False, pin_factory=None*)

Extends [LEDBoard](#) (page 113) for the [Ryanteck SnowPi](#)⁴⁰⁰ board.

The SnowPi pins are fixed and therefore there's no need to specify them when constructing this class. The following example turns on the eyes, sets the nose pulsing, and the arms blinking:

```
from gpiozero import SnowPi

snowman = SnowPi(pwm=True)
snowman.eyes.on()
snowman.nose.pulse()
snowman.arms.blink()
```

Parameters

- **pwm** (*bool*⁴⁰¹) – If True, construct [PWMLed](#) (page 88) instances to represent each LED. If False (the default), construct regular [LED](#) (page 87) instances.
- **initial_value** (*bool*⁴⁰²) – 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 180)) – See [API - Pins](#) (page 177) 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*⁴⁰³) – Number of seconds on. Defaults to 1 second.
- **off_time** (*float*⁴⁰⁴) – Number of seconds off. Defaults to 1 second.
- **fade_in_time** (*float*⁴⁰⁵) – Number of seconds to spend fading in. Defaults to 0. Must be 0 if *pwm* was False when the class was constructed ([ValueError](#)⁴⁰⁶ will be raised if not).
- **fade_out_time** (*float*⁴⁰⁷) – Number of seconds to spend fading out. Defaults to 0. Must be 0 if *pwm* was False when the class was constructed ([ValueError](#)⁴⁰⁸ will be raised if not).
- **n** (*int*⁴⁰⁹) – Number of times to blink; None (the default) means forever.

⁴⁰⁰ <https://ryanteck.uk/raspberry-pi/114-snowpi-the-gpio-snowman-for-raspberry-pi-0635648608303.html>

⁴⁰¹ <https://docs.python.org/3.5/library/functions.html#bool>

⁴⁰² <https://docs.python.org/3.5/library/functions.html#bool>

⁴⁰³ <https://docs.python.org/3.5/library/functions.html#float>

⁴⁰⁴ <https://docs.python.org/3.5/library/functions.html#float>

⁴⁰⁵ <https://docs.python.org/3.5/library/functions.html#float>

⁴⁰⁶ <https://docs.python.org/3.5/library/exceptions.html#ValueError>

⁴⁰⁷ <https://docs.python.org/3.5/library/functions.html#float>

⁴⁰⁸ <https://docs.python.org/3.5/library/exceptions.html#ValueError>

⁴⁰⁹ <https://docs.python.org/3.5/library/functions.html#int>

- **background** (*bool*⁴¹⁰) – 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 161) descendants can also be used as context managers using the `with`⁴¹¹ 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*⁴¹²) – Number of seconds to spend fading in. Defaults to 1.
- **fade_out_time** (*float*⁴¹³) – Number of seconds to spend fading out. Defaults to 1.
- **n** (*int*⁴¹⁴) – Number of times to blink; None (the default) means forever.
- **background** (*bool*⁴¹⁵) – 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).

⁴¹⁰ <https://docs.python.org/3.5/library/functions.html#bool>

⁴¹¹ https://docs.python.org/3.5/reference/compound_stmts.html#with

⁴¹² <https://docs.python.org/3.5/library/functions.html#float>

⁴¹³ <https://docs.python.org/3.5/library/functions.html#float>

⁴¹⁴ <https://docs.python.org/3.5/library/functions.html#int>

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

toggle (*args)

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

closed

Returns `True` if the device is closed (see the `close()` (page 150) 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 151). Unlike `value` (page 151), this is *always* a boolean.

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 151).

source_delay

The delay (measured in seconds) in the loop used to read values from `source` (page 151). Defaults to 0.01 seconds which is generally sufficient to keep CPU usage to a minimum while providing adequate responsiveness.

value

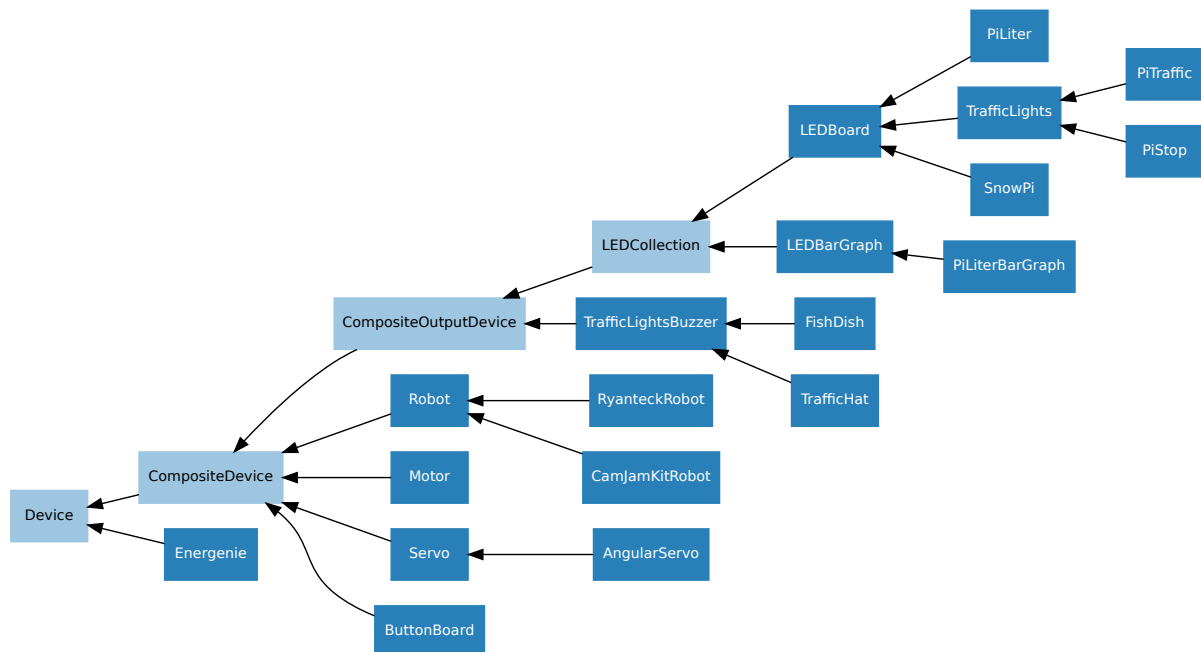
A tuple containing a value for each subordinate device. This property can also be set to update the state of all subordinate output devices.

values

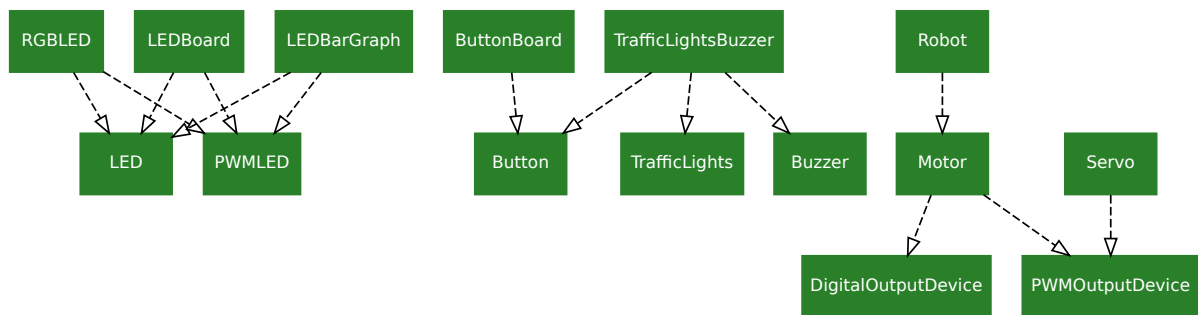
An infinite iterator of values read from `value`.

15.22 Base Classes

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



For composite devices, the following chart shows which devices are composed of which other devices:



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

15.23 LEDCollection

class gpiozero.LEDCollection(*pins, pwm=False, active_high=True, initial_value=False, pin_factory=None, **named_pins)
 Extends [CompositeOutputDevice](#) (page 152). Abstract base class for [LEDBoard](#) (page 113) and [LEDBarGraph](#) (page 116).

leds

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

15.24 CompositeOutputDevice

class gpiozero.CompositeOutputDevice(*args, _order=None, pin_factory=None, **kwargs)
 Extends [CompositeDevice](#) (page 153) with [on\(\)](#) (page 152), [off\(\)](#) (page 152), and [toggle\(\)](#) (page 152) methods for controlling subordinate output devices. Also extends [value](#) (page 152) to be writeable.

Parameters

- **_order** ([list](#)⁴¹⁶) – If specified, this is the order of named items specified by keyword arguments (to ensure that the [value](#) (page 152) 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 180)) – See [API - Pins](#) (page 177) 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.

⁴¹⁶ <https://docs.python.org/3.5/library/stdtypes.html#list>

15.25 CompositeDevice

class gpiozero.**CompositeDevice** (*args, _order=None, pin_factory=None, **kwargs)

Extends *Device* (page 161). 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*⁴¹⁷) – If specified, this is the order of named items specified by keyword arguments (to ensure that the *value* (page 153) tuple is constructed with a specific order). All keyword arguments *must* be included in the collection. If omitted, an alphabetically sorted order will be selected for keyword arguments.

close()

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

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

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

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

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

Device (page 161) descendants can also be used as context managers using the `with`⁴¹⁸ 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 153) 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 153). Unlike *value* (page 153), 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.

⁴¹⁷ <https://docs.python.org/3.5/library/stdtypes.html#list>

⁴¹⁸ 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!

16.1 TimeOfDay

class gpiozero.**TimeOfDay** (*start_time*, *end_time*, *utc=True*)

Extends *InternalDevice* (page 157) 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`⁴¹⁹ instances.

The following example turns on a lamp attached to an *Energenie* (page 144) 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`⁴²⁰) – The time from which the device will be considered active.
- **end_time** (`time`⁴²¹) – The time after which the device will be considered inactive.

⁴¹⁹ <https://docs.python.org/3.5/library/datetime.html#datetime.time>

⁴²⁰ <https://docs.python.org/3.5/library/datetime.html#datetime.time>

⁴²¹ <https://docs.python.org/3.5/library/datetime.html#datetime.time>

- **utc** (*bool*⁴²²) – If `True` (the default), a naive UTC time will be used for the comparison rather than a local time-zone reading.

16.2 PingServer

class `gpiozero.PingServer` (*host*)

Extends *InternalDevice* (page 157) 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 162) 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*⁴²³) – The hostname or IP address to attempt to ping.

16.3 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 157) 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*⁴²⁴) – The file from which to read the temperature. This defaults to the sysfs file `/sys/class/thermal/thermal_zone0/temp`. Whatever file is specified is expected to contain a single line containing the temperature in milli-degrees celsius.

⁴²² <https://docs.python.org/3.5/library/functions.html#bool>

⁴²³ <https://docs.python.org/3.5/library/stdtypes.html#str>

⁴²⁴ <https://docs.python.org/3.5/library/stdtypes.html#str>

- **min_temp** (*float*⁴²⁵) – The temperature at which `value` will read 0.0. This defaults to 0.0.
- **max_temp** (*float*⁴²⁶) – The temperature at which `value` will read 1.0. This defaults to 100.0.
- **threshold** (*float*⁴²⁷) – The temperature above which the device will be considered “active”. This defaults to 80.0.

is_active

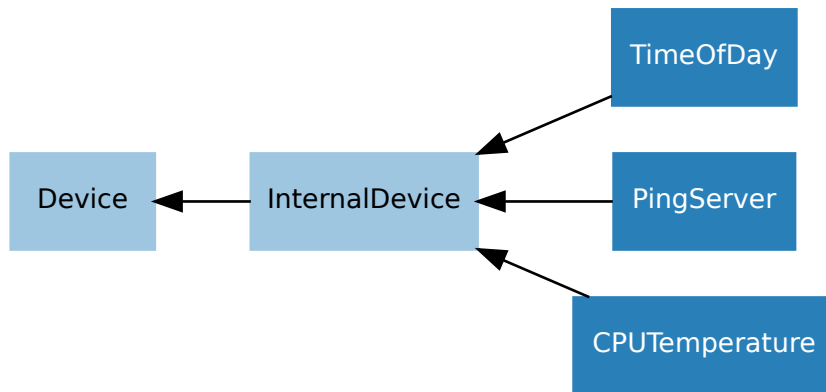
Returns `True` when the CPU *temperature* (page 157) exceeds the `threshold`.

temperature

Returns the current CPU temperature in degrees celsius.

16.4 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.

16.5 InternalDevice

class `gpiozero.InternalDevice`

Extends *Device* (page 161) 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.

⁴²⁵ <https://docs.python.org/3.5/library/functions.html#float>

⁴²⁶ <https://docs.python.org/3.5/library/functions.html#float>

⁴²⁷ <https://docs.python.org/3.5/library/functions.html#float>

CHAPTER 17

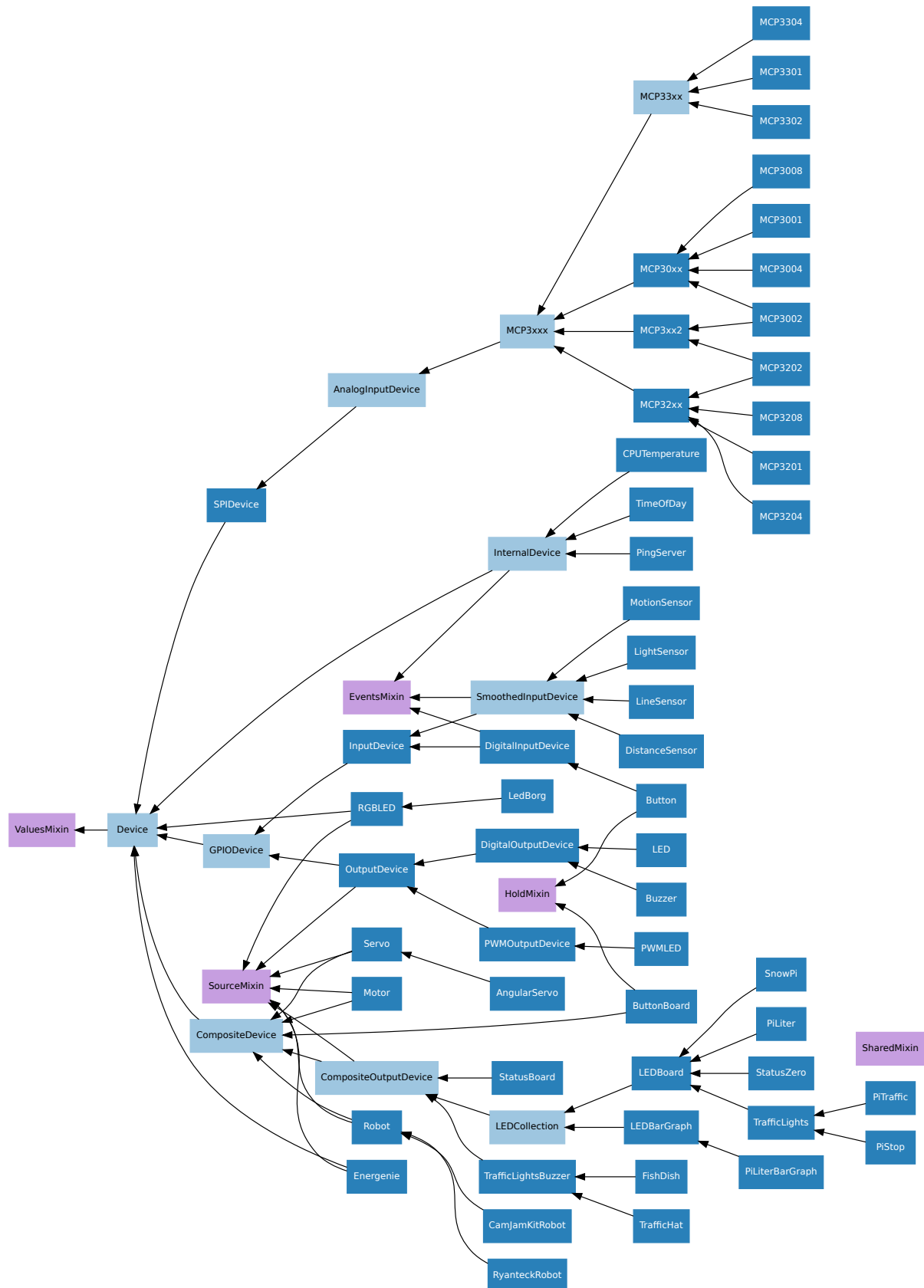
API - Generic Classes

The GPIO Zero class hierarchy is quite extensive. It contains several base classes (most of which are documented in their corresponding chapters):

- *Device* (page 161) is the root of the hierarchy, implementing base functionality like `close()` (page 161) and context manager handlers.
- *GPIODevice* (page 84) represents individual devices that attach to a single GPIO pin
- *SPIDevice* (page 111) represents devices that communicate over an SPI interface (implemented as four GPIO pins)
- *InternalDevice* (page 157) represents devices that are entirely internal to the Pi (usually operating system related services)
- *CompositeDevice* (page 153) represents devices composed of multiple other devices like HATs

There are also several *mixin classes*⁴²⁸ 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):

⁴²⁸ <https://en.wikipedia.org/wiki/Mixin>



17.1 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 161) property, the *value* (page 161) property, and the *close()* (page 161) 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 161) descendents can also be used as context managers using the *with*⁴²⁹ 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 161) 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 161). Unlike *value* (page 161), 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.

17.2 ValuesMixin

class gpiozero.ValuesMixin (...)

Adds a *values* (page 162) property to the class which returns an infinite generator of readings from the

⁴²⁹ 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*.

17.3 SourceMixin

class `gpiozero.SourceMixin(...)`

Adds a *source* (page 162) 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 162). Defaults to 0.01 seconds which is generally sufficient to keep CPU usage to a minimum while providing adequate responsiveness.

17.4 SharedMixin

class `gpiozero.SharedMixin(...)`

This mixin marks a class as “shared”. In this case, the meta-class (GPIONMeta) will use `__shared_key()` (page 162) 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.

17.5 EventsMixin

class `gpiozero.EventsMixin(...)`

Adds edge-detected `when_activated()` (page 163) and `when_deactivated()` (page 163) events to a device based on changes to the `is_active` (page 161) property common to all devices. Also adds `wait_for_active()` (page 162) and `wait_for_inactive()` (page 163) 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*⁴³⁰) – 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*⁴³¹) – 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.

17.6 HoldMixin

class gpiozero.HoldMixin(...)

Extends *EventsMixin* (page 162) to add the *when_held* (page 163) event and the machinery to fire that event repeatedly (when *hold_repeat* (page 163) is *True*) at intervals defined by *hold_time* (page 163).

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 163) event rather than when the device activated, in contrast to *active_time* (page 163). If the device is not currently held, this is *None*.

hold_repeat

If *True*, *when_held* (page 163) will be executed repeatedly with *hold_time* (page 163) 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 163) handler. If *hold_repeat* (page 163) is *True*, this is also the length of time between invocations of *when_held* (page 163).

⁴³⁰ <https://docs.python.org/3.5/library/functions.html#float>

⁴³¹ <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 163) seconds.

when_held

The function to run when the device has remained active for *hold_time* (page 163) 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 18

API - Device Source Tools

GPIO Zero includes several utility routines which are intended to be used with the *Source/Values* (page 51) 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 162) and *values* (page 162) deal with infinite iterators, another excellent source of utilities is the `itertools`⁴³² 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!

18.1 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*

⁴³² <https://docs.python.org/3.5/library/itertools.html#module-itertools>

can optionally be used to add [hysteresis](https://en.wikipedia.org/wiki/Hysteresis)⁴³³ 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*.

⁴³³ <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 167), *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)
```

(continues on next page)

(continued from previous page)

```

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).

18.2 Combining sources

`gpiozero.tools.all_values` (**values*)

Returns the *logical conjunction*⁴³⁴ 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)

```

(continues on next page)

⁴³⁴ https://en.wikipedia.org/wiki/Logical_conjunction

(continued from previous page)

```
btn2 = Button(21)
led.source = all_values(btn1.values, btn2.values)
pause()
```

`gpiozero.tools.any_values(*values)`

Returns the [logical disjunction](#)⁴³⁵ 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
```

(continues on next page)

⁴³⁵ https://en.wikipedia.org/wiki/Logical_disjunction

(continued from previous page)

```
from signal import pause

pot1 = MCP3008(channel=0)
pot2 = MCP3008(channel=1)
pot3 = MCP3008(channel=2)
led = PWMLED(4)

led.source = scaled(summed(pot1.values, pot2.values, pot3.values), 0, 1, 0, 3)

pause()
```

18.3 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 168).

`gpiozero.tools.ramping_values` (*period=360*)

Provides an infinite source of values representing a triangle wave (from 0 to 1 and back again) which repeats every *period* values. For example, to pulse an LED once a second:

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

red = PWMLED(2)
```

(continues on next page)

(continued from previous page)

```
red.source_delay = 0.01
red.source = ramping_values(100)

pause()
```

If you require a wider range than 0 to 1, see [scaled\(\)](#) (page 168).

`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 168).

`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 168).

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 173) instance containing information about a *revision* of the Raspberry Pi.

Parameters *revision* (*str*⁴³⁶) – 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*⁴³⁷ 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 codes⁴³⁸. 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 193). If multiple pins support the desired *function*,

⁴³⁶ <https://docs.python.org/3.5/library/stdtypes.html#str>

⁴³⁷ <https://docs.python.org/3.5/library/collections.html#collections.namedtuple>

⁴³⁸ https://en.wikipedia.org/wiki/ANSI_escape_code

PinMultiplePins (page 193) will be raised (use *physical_pins()* (page 174) if you expect multiple pins in the result, such as for electrical ground).

Parameters *function* (*str*⁴³⁹) – 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*⁴⁴⁰ which is not indexable. Use *physical_pin()* (page 173) if you are expecting a single return value.

Parameters *function* (*str*⁴⁴¹) – 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 193) or *PinMultiplePins* (page 193) may be raised if the function is not associated with a single pin.

Parameters *function* (*str*⁴⁴²) – 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*⁴⁴³) 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.

⁴³⁹ <https://docs.python.org/3.5/library/stdtypes.html#str>

⁴⁴⁰ <https://docs.python.org/3.5/library/stdtypes.html#set>

⁴⁴¹ <https://docs.python.org/3.5/library/stdtypes.html#str>

⁴⁴² <https://docs.python.org/3.5/library/stdtypes.html#str>

⁴⁴³ 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 175) 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 173) 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 173) object in a format string. For example:

```
from gpiozero import *
print('{0:board}'.format(pi_info()))
```

class gpiozero.HeaderInfo

This class is a `namedtuple()` ⁴⁴⁴ 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']))
```

⁴⁴⁴ <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 codes⁴⁴⁵. 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 176) tuples.

class gpiozero.PinInfo

This class is a *namedtuple()*⁴⁴⁶ 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)

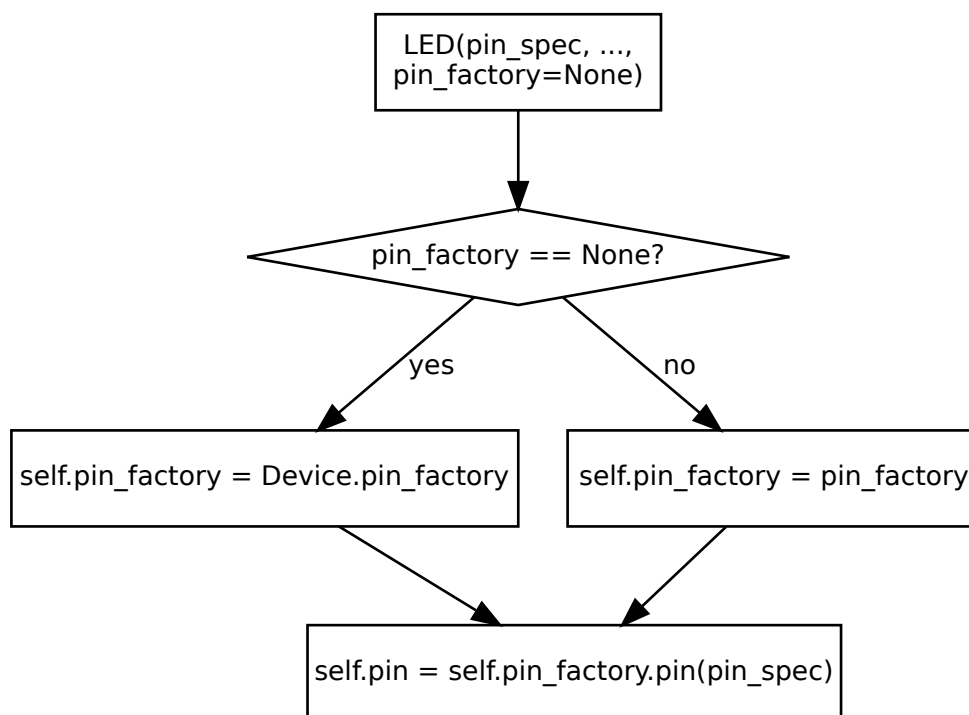
⁴⁴⁵ https://en.wikipedia.org/wiki/ANSI_escape_code

⁴⁴⁶ <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 180) which turns it into a *Pin* (page 181) implementation. The default factory can be queried (and changed) with `Device.pin_factory`, i.e. the `pin_factory` attribute of the *Device* (page 161) 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`⁴⁴⁷ will be raised.

20.1 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 180) and *Pin* (page 181) 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> (page 187)	<code>gpiozero.pins.rpigpio.RPiGPIOPin</code> (page 188)
rpio	<code>gpiozero.pins.rpio.RPIOFactory</code> (page 188)	<code>gpiozero.pins.rpio.RPIOPin</code> (page 188)
pig-pio	<code>gpiozero.pins.pigpio.PiGPIOFactory</code> (page 188)	<code>gpiozero.pins.pigpio.PiGPIOPin</code> (page 189)
native	<code>gpiozero.pins.native.NativeFactory</code> (page 189)	<code>gpiozero.pins.native.NativePin</code> (page 189)

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:

⁴⁴⁷ <https://docs.python.org/3.5/library/exceptions.html#ImportError>

```

from gpiozero.pins.native import NativeFactory
from gpiozero import Device, LED

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 189) 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).

20.2 Mock pins

There's also a `gpiozero.pins.mock.MockFactory` (page 190) 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

```

(continues on next page)

(continued from previous page)

```
# 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.

20.3 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 180)
- `reserve_pins()` (page 180)
- `release_pins()` (page 180)
- `release_all()` (page 180)
- `pin()` (page 180)
- `spi()` (page 181)
- `_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 181) 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 180) 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 180) for further information).

release_pins (*reserver*, **pins*)

Releases the reservation of *reserver* against *pins*. This is typically called during `Device.close()` (page 161) 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 180).

spi (***spi_args*)

Returns an instance of an *SPI* (page 183) 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 192).

pi_info

Returns a *PiBoardInfo* (page 173) 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 181)
- `output_with_state()` (page 182)
- `input_with_pull()` (page 181)
- `_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 181) 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, descendants 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:

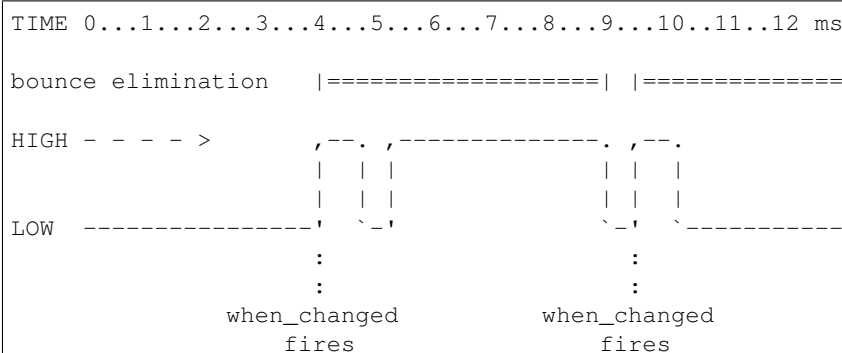
```
pin.function = 'output'
pin.state = state
```

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`.

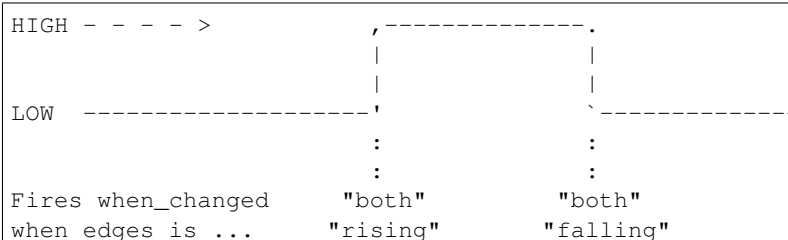
For example, if *edges* (page 182) is currently “rising”, *bounce* (page 182) is currently 5/1000 (5ms), then the waveform below will only fire *when_changed* (page 183) on two occasions despite there being three rising edges:



If the pin does not support edge detection, attempts to set this property will raise *PinEdgeDetectUnsupported* (page 193). 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 183). 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 193).

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 193) 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 193) 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 193). If the specified value is not supported by the underlying hardware, *PinInvalidPull* (page 193) 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 183) 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 182) 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 193) will be raised. If an invalid value is specified for this attribute, *PinInvalidState* (page 193) 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 182) 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 193).

class gpiozero.**SPI**

Abstract interface for *Serial Peripheral Interface*⁴⁴⁸ (SPI) implementations. Descendents *must* override the following methods:

- *transfer()* (page 184)
- *_get_clock_mode()*

Descendents *may* override the following methods, if applicable:

- *read()* (page 184)
- *write()* (page 184)
- *_set_clock_mode()*
- *_get_lsb_first()*
- *_set_lsb_first()*
- *_get_select_high()*
- *_set_select_high()*
- *_get_bits_per_word()*

⁴⁴⁸ https://en.wikipedia.org/wiki/Serial_Peripheral_Interface_Bus

- `_set_bits_per_word()`

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 184) of the interface.

This method is typically used with read-only devices that feature half-duplex communication. See `transfer()` (page 184) 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 184) 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 184) 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 184) 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 184) 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 185) and `clock_phase` (page 184) attributes combined according to the following table:

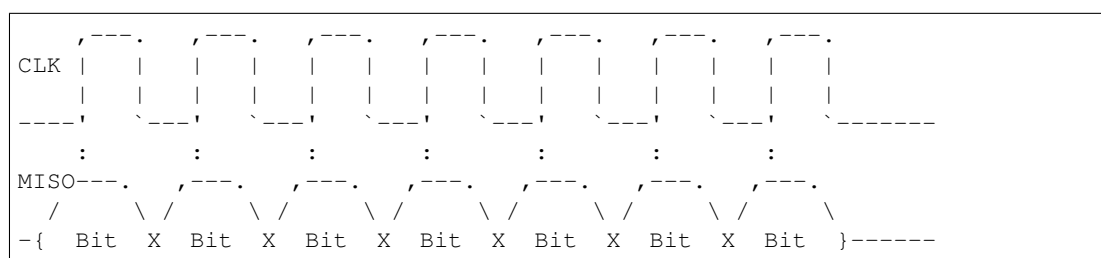
mode	polarity (CPOL)	phase (CPHA)
0	False	False
1	False	True
2	True	False
3	True	True

Adjusting this value adjusts both the `clock_polarity` (page 185) and `clock_phase` (page 184) attributes simultaneously.

clock_phase

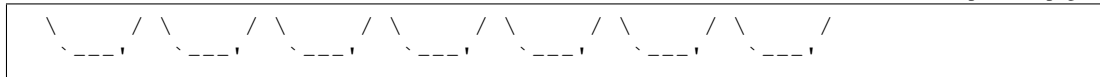
The phase of the SPI clock pin. If this is `False` (the default), data will be read from the MISO pin when the clock pin activates. Setting this to `True` will cause data to be read from the MISO pin when the clock pin deactivates. On many data sheets this is documented as the CPHA value. Whether the clock edge is rising or falling when the clock is considered activated is controlled by the `clock_polarity` (page 185) attribute (corresponding to CPOL).

The following diagram indicates when data is read when `clock_polarity` (page 185) is `False`, and `clock_phase` (page 184) is `False` (the default), equivalent to CPHA 0:

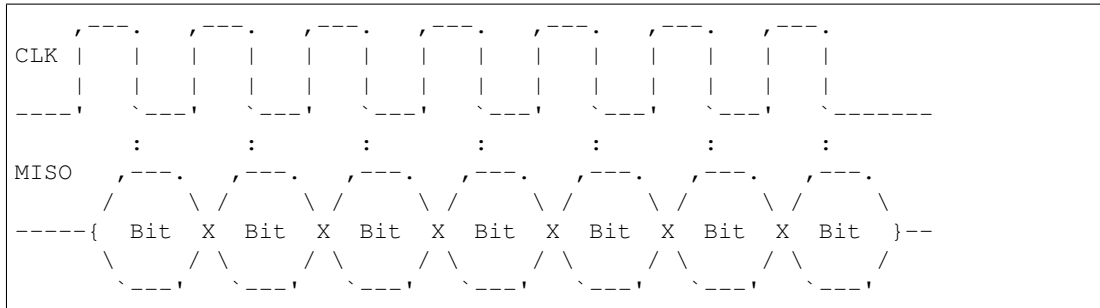


(continues on next page)

(continued from previous page)



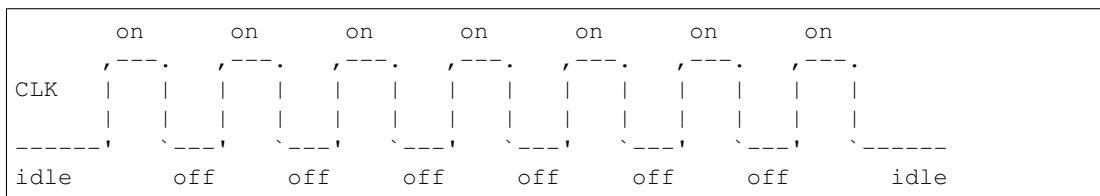
The following diagram indicates when data is read when `clock_polarity` (page 185) is `False`, but `clock_phase` (page 184) is `True`, equivalent to CPHA 1:



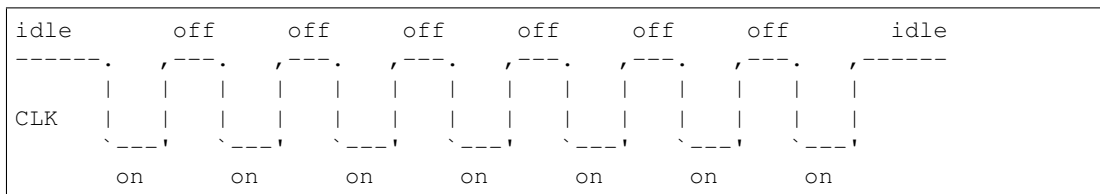
`clock_polarity`

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 185) is `False` (the default), equivalent to CPOL 0:



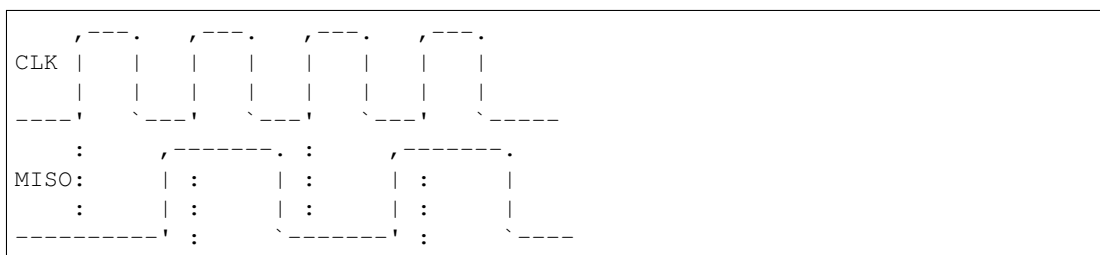
The following diagram illustrates the waveform when `clock_polarity` (page 185) is `True`, equivalent to CPOL 1:



`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)⁴⁴⁹ of the connection.

The following diagram shows the a word containing the number 5 (binary 0101) transmitted on MISO with `bits_per_word` (page 184) set to 4, and `clock_mode` (page 184) set to 0, when `lsb_first` (page 185) is `False` (the default):



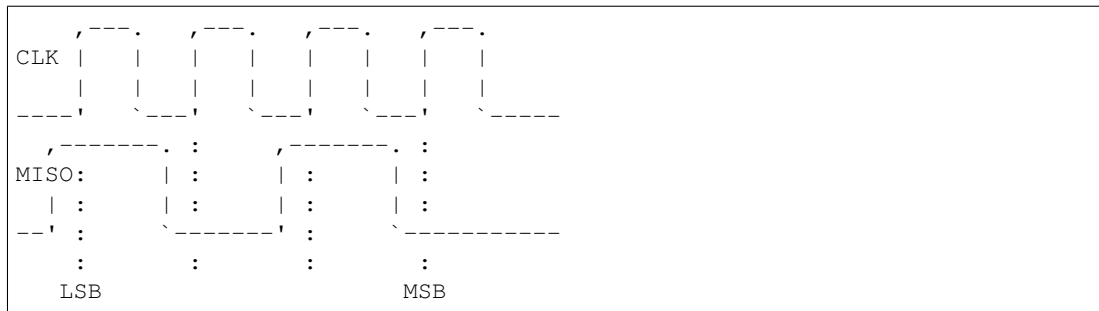
(continues on next page)

⁴⁴⁹ https://en.wikipedia.org/wiki/Endianness#Bit_endianness

(continued from previous page)



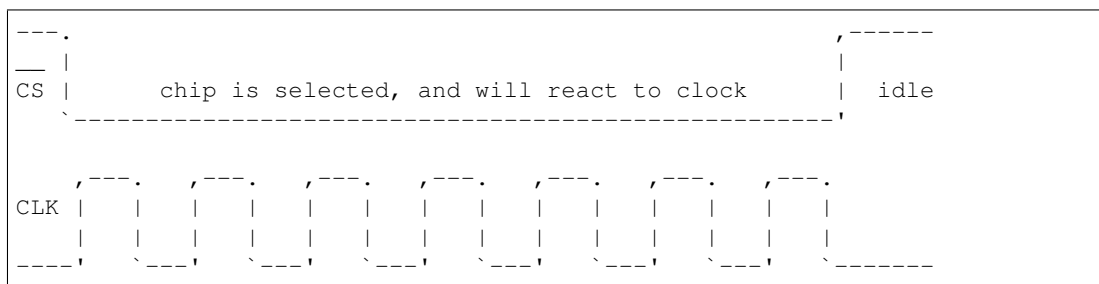
And now with `lsb_first` (page 185) set to `True` (and all other parameters the same):



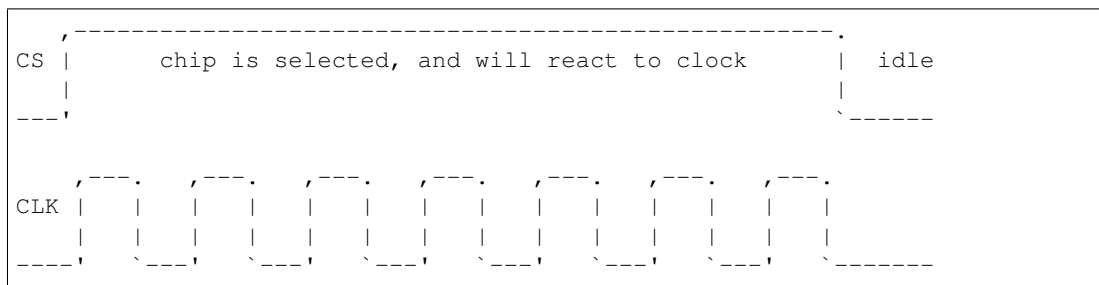
`select_high`

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 185) is `False`, and `select_high` (page 186) is `False` (the default):



And when `select_high` (page 186) is `True`:



`class gpiozero.pins.pi.PiFactory`

Abstract base class representing hardware attached to a Raspberry Pi. This forms the base of `LocalPiFactory` (page 187).

`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` 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 186) for the same pin specification must return the same object.

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 181). 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 188), `RPiOPin` (page 188), and `NativePin` (page 189)).

class `gpiozero.pins.local.LocalPiPin` (*factory, number*)

Abstract base class representing a multi-function GPIO pin attached to the local Raspberry Pi.

20.4 RPi.GPIO

class `gpiozero.pins.rpigpio.RPiGPIOFactory`

Uses the `RPi.GPIO`⁴⁵⁰ library to interface to the Pi's GPIO pins. This is the default pin implementation if

⁴⁵⁰ <https://pypi.python.org/pypi/RPi.GPIO>

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)
```

class `gpiozero.pins.rpigpio.RPiGPIOPin` (*factory, number*)

Pin implementation for the [RPi.GPIO](#)⁴⁵¹ library. See [RPiGPIOFactory](#) (page 187) for more information.

20.5 RPIO

class `gpiozero.pins.rpio.RPIOFactory`

Uses the [RPIO](#)⁴⁵² 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](#)⁴⁵³ library. See [RPIOFactory](#) (page 188) for more information.

20.6 PiGPIO

class `gpiozero.pins.pigpio.PiGPIOFactory` (*host='localhost', port=8888*)

Uses the [pigpio](#)⁴⁵⁴ 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](#)⁴⁵⁵ client library installed on it)

⁴⁵¹ <https://pypi.python.org/pypi/RPi.GPIO>

⁴⁵² <https://pythonhosted.org/RPIO/>

⁴⁵³ <https://pythonhosted.org/RPIO/>

⁴⁵⁴ <http://abyz.co.uk/rpi/pigpio/>

⁴⁵⁵ <http://abyz.co.uk/rpi/pigpio/>

- 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:

```
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`⁴⁵⁶ library. See `PiGPIOFactory` (page 188) for more information.

20.7 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 189) for more information.

⁴⁵⁶ <http://abyz.co.uk/rpi/pigpio/>

20.8 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 190) 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 190) 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 190) 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.

class gpiozero.pins.mock.**MockPin** (*factory, number*)

A mock pin used primarily for testing. This class does *not* support PWM.

close ()

Cleans up the resources allocated to the pin. After this method is called, this `Pin` instance may no longer be used to query or control the pin's state.

class gpiozero.pins.mock.**MockPWMPin** (*factory, number*)

This derivative of `MockPin` (page 190) adds PWM support.

class gpiozero.pins.mock.**MockConnectedPin** (*factory, number, input_pin=None*)

This derivative of `MockPin` (page 190) 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 190) 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 190) is intended to be used with another `MockPin` (page 190) 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 21

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 191), 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 103) you would expect a *ValueError*⁴⁵⁷ to be raised. In fact, a *OutputDeviceBadValue* (page 192) error will be raised. However, note that this descends from both *GPIOZeroError* (page 191) (indirectly) and from *ValueError*⁴⁵⁸ so you can still do:

```
from gpiozero import *

led = PWMLED(17)
try:
    led.value = 2
except ValueError:
    print('Bad value specified')
```

21.1 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

⁴⁵⁷ <https://docs.python.org/3.5/library/exceptions.html#ValueError>

⁴⁵⁸ <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 144)

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 111)

exception `gpiozero.SPIBadChannel`

Error raised when an invalid channel is given to an [AnalogInputDevice](#) (page 110)

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 191)

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

21.2 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

22.1 Release 1.4.1 (2018-02-20)

This release is mostly bug-fixes, but a few enhancements have made it in too:

- Added `curve_left` and `curve_right` parameters to `Robot.forward()` (page 137) and `Robot.backward()` (page 137).([#306](#)⁴⁵⁹ and [#619](#)⁴⁶⁰)
- Fixed `DistanceSensor` (page 79) returning incorrect readings after a long pause, and added a lock to ensure multiple distance sensors can operate simultaneously in a single project ([#584](#)⁴⁶¹, [#595](#)⁴⁶², [#617](#)⁴⁶³, [#618](#)⁴⁶⁴)
- Added support for phase/enable motor drivers with `PhaseEnableMotor` (page 94), `PhaseEnableRobot` (page 138), and descendants, thanks to Ian Harcombe! ([#386](#)⁴⁶⁵)
- A variety of other minor enhancements, largely thanks to Andrew Scheller! ([#479](#)⁴⁶⁶, [#489](#)⁴⁶⁷, [#491](#)⁴⁶⁸, [#492](#)⁴⁶⁹)

22.2 Release 1.4.0 (2017-07-26)

- Pin factory is now *configurable from device constructors* (page 178) 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](#)⁴⁷⁰)

⁴⁵⁹ <https://github.com/RPi-Distro/python-gpiozero/issues/306>

⁴⁶⁰ <https://github.com/RPi-Distro/python-gpiozero/issues/619>

⁴⁶¹ <https://github.com/RPi-Distro/python-gpiozero/issues/584>

⁴⁶² <https://github.com/RPi-Distro/python-gpiozero/issues/595>

⁴⁶³ <https://github.com/RPi-Distro/python-gpiozero/issues/617>

⁴⁶⁴ <https://github.com/RPi-Distro/python-gpiozero/issues/618>

⁴⁶⁵ <https://github.com/RPi-Distro/python-gpiozero/issues/386>

⁴⁶⁶ <https://github.com/RPi-Distro/python-gpiozero/issues/479>

⁴⁶⁷ <https://github.com/RPi-Distro/python-gpiozero/issues/489>

⁴⁶⁸ <https://github.com/RPi-Distro/python-gpiozero/issues/491>

⁴⁶⁹ <https://github.com/RPi-Distro/python-gpiozero/issues/492>

⁴⁷⁰ <https://github.com/RPi-Distro/python-gpiozero/issues/279>

- Major work on SPI, primarily to support remote hardware SPI ([#421](#)⁴⁷¹, [#459](#)⁴⁷², [#465](#)⁴⁷³, [#468](#)⁴⁷⁴, [#575](#)⁴⁷⁵)
- Pin reservation now works properly between GPIO and SPI devices ([#459](#)⁴⁷⁶, [#468](#)⁴⁷⁷)
- Lots of work on the documentation: *source/values chapter* (page 51), better charts, more recipes, *remote GPIO configuration* (page 35), mock pins, better PDF output ([#484](#)⁴⁷⁸, [#469](#)⁴⁷⁹, [#523](#)⁴⁸⁰, [#520](#)⁴⁸¹, [#434](#)⁴⁸², [#565](#)⁴⁸³, [#576](#)⁴⁸⁴)
- Support for *StatusZero* (page 145) and *StatusBoard* (page 148) HATs ([#558](#)⁴⁸⁵)
- Added **pinout** command line tool to provide a simple reference to the GPIO layout and information about the associated Pi ([#497](#)⁴⁸⁶, [#504](#)⁴⁸⁷) thanks to Stewart Adcock for the initial work
- `pi_info()` (page 173) made more lenient for new (unknown) Pi models ([#529](#)⁴⁸⁸)
- Fixed a variety of packaging issues ([#535](#)⁴⁸⁹, [#518](#)⁴⁹⁰, [#519](#)⁴⁹¹)
- Improved text in factory fallback warnings ([#572](#)⁴⁹²)

22.3 Release 1.3.2 (2017-03-03)

- Added new Pi models to stop `pi_info()` (page 173) breaking
- Fix issue with `pi_info()` (page 173) breaking on unknown Pi models

22.4 Release 1.3.1 (2016-08-31 ... later)

- Fixed hardware SPI support which Dave broke in 1.3.0. Sorry!
- Some minor docs changes

22.5 Release 1.3.0 (2016-08-31)

- Added *ButtonBoard* (page 117) for reading multiple buttons in a single class ([#340](#)⁴⁹³)
- Added *Servo* (page 95) and *AngularServo* (page 96) classes for controlling simple servo motors ([#248](#)⁴⁹⁴)

⁴⁷¹ <https://github.com/RPi-Distro/python-gpiozero/issues/421>

⁴⁷² <https://github.com/RPi-Distro/python-gpiozero/issues/459>

⁴⁷³ <https://github.com/RPi-Distro/python-gpiozero/issues/465>

⁴⁷⁴ <https://github.com/RPi-Distro/python-gpiozero/issues/468>

⁴⁷⁵ <https://github.com/RPi-Distro/python-gpiozero/issues/575>

⁴⁷⁶ <https://github.com/RPi-Distro/python-gpiozero/issues/459>

⁴⁷⁷ <https://github.com/RPi-Distro/python-gpiozero/issues/468>

⁴⁷⁸ <https://github.com/RPi-Distro/python-gpiozero/issues/484>

⁴⁷⁹ <https://github.com/RPi-Distro/python-gpiozero/issues/469>

⁴⁸⁰ <https://github.com/RPi-Distro/python-gpiozero/issues/523>

⁴⁸¹ <https://github.com/RPi-Distro/python-gpiozero/issues/520>

⁴⁸² <https://github.com/RPi-Distro/python-gpiozero/issues/434>

⁴⁸³ <https://github.com/RPi-Distro/python-gpiozero/issues/565>

⁴⁸⁴ <https://github.com/RPi-Distro/python-gpiozero/issues/576>

⁴⁸⁵ <https://github.com/RPi-Distro/python-gpiozero/issues/558>

⁴⁸⁶ <https://github.com/RPi-Distro/python-gpiozero/issues/497>

⁴⁸⁷ <https://github.com/RPi-Distro/python-gpiozero/issues/504>

⁴⁸⁸ <https://github.com/RPi-Distro/python-gpiozero/issues/529>

⁴⁸⁹ <https://github.com/RPi-Distro/python-gpiozero/issues/535>

⁴⁹⁰ <https://github.com/RPi-Distro/python-gpiozero/issues/518>

⁴⁹¹ <https://github.com/RPi-Distro/python-gpiozero/issues/519>

⁴⁹² <https://github.com/RPi-Distro/python-gpiozero/issues/572>

⁴⁹³ <https://github.com/RPi-Distro/python-gpiozero/issues/340>

⁴⁹⁴ <https://github.com/RPi-Distro/python-gpiozero/issues/248>

- Lots of work on supporting easier use of internal and third-party pin implementations (#359⁴⁹⁵)
- *Robot* (page 136) now has a proper *value* (page 138) attribute (#305⁴⁹⁶)
- Added *CPUTemperature* (page 156) as another demo of “internal” devices (#294⁴⁹⁷)
- A temporary work-around for an issue with *DistanceSensor* (page 79) was included but a full fix is in the works (#385⁴⁹⁸)
- More work on the documentation (#320⁴⁹⁹, #295⁵⁰⁰, #289⁵⁰¹, 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!

22.6 Release 1.2.0 (2016-04-10)

- Added *Energenie* (page 144) class for controlling Energenie plugs (#69⁵⁰²)
- Added *LineSensor* (page 75) class for single line-sensors (#109⁵⁰³)
- Added *DistanceSensor* (page 79) class for HC-SR04 ultra-sonic sensors (#114⁵⁰⁴)
- Added *SnowPi* (page 149) class for the Ryantek Snow-pi board (#130⁵⁰⁵)
- Added *when_held* (page 74) (and related properties) to *Button* (page 73) (#115⁵⁰⁶)
- Fixed issues with installing GPIO Zero for python 3 on Raspbian Wheezy releases (#140⁵⁰⁷)
- Added support for lots of ADC chips (MCP3xxx family) (#162⁵⁰⁸) - many thanks to pcopa and lurch!
- Added support for pigpiod as a pin implementation with *PiGPIOPin* (page 189) (#180⁵⁰⁹)
- Many refinements to the base classes mean more consistency in composite devices and several bugs squashed (#164⁵¹⁰, #175⁵¹¹, #182⁵¹², #189⁵¹³, #193⁵¹⁴, #229⁵¹⁵)
- GPIO Zero is now aware of what sort of Pi it’s running on via *pi_info()* (page 173) 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) (#222⁵¹⁶)
- The source/values system was enhanced to ensure normal usage doesn’t stress the CPU and lots of utilities were added (#181⁵¹⁷, #251⁵¹⁸)

⁴⁹⁵ <https://github.com/RPi-Distro/python-gpiozero/issues/359>

⁴⁹⁶ <https://github.com/RPi-Distro/python-gpiozero/issues/305>

⁴⁹⁷ <https://github.com/RPi-Distro/python-gpiozero/issues/294>

⁴⁹⁸ <https://github.com/RPi-Distro/python-gpiozero/issues/385>

⁴⁹⁹ <https://github.com/RPi-Distro/python-gpiozero/issues/320>

⁵⁰⁰ <https://github.com/RPi-Distro/python-gpiozero/issues/295>

⁵⁰¹ <https://github.com/RPi-Distro/python-gpiozero/issues/289>

⁵⁰² <https://github.com/RPi-Distro/python-gpiozero/issues/69>

⁵⁰³ <https://github.com/RPi-Distro/python-gpiozero/issues/109>

⁵⁰⁴ <https://github.com/RPi-Distro/python-gpiozero/issues/114>

⁵⁰⁵ <https://github.com/RPi-Distro/python-gpiozero/issues/130>

⁵⁰⁶ <https://github.com/RPi-Distro/python-gpiozero/issues/115>

⁵⁰⁷ <https://github.com/RPi-Distro/python-gpiozero/issues/140>

⁵⁰⁸ <https://github.com/RPi-Distro/python-gpiozero/issues/162>

⁵⁰⁹ <https://github.com/RPi-Distro/python-gpiozero/issues/180>

⁵¹⁰ <https://github.com/RPi-Distro/python-gpiozero/issues/164>

⁵¹¹ <https://github.com/RPi-Distro/python-gpiozero/issues/175>

⁵¹² <https://github.com/RPi-Distro/python-gpiozero/issues/182>

⁵¹³ <https://github.com/RPi-Distro/python-gpiozero/issues/189>

⁵¹⁴ <https://github.com/RPi-Distro/python-gpiozero/issues/193>

⁵¹⁵ <https://github.com/RPi-Distro/python-gpiozero/issues/229>

⁵¹⁶ <https://github.com/RPi-Distro/python-gpiozero/issues/222>

⁵¹⁷ <https://github.com/RPi-Distro/python-gpiozero/issues/181>

⁵¹⁸ <https://github.com/RPi-Distro/python-gpiozero/issues/251>

And I'll just add a note of thanks to the many people in the community who contributed to this release: we've had some great PRs, suggestions, and bug reports in this version. Of particular note:

- Schelto van Doorn was instrumental in adding support for numerous ADC chips
- Alex Eames generously donated a RasPiO Analog board which was extremely useful in developing the software SPI interface (and testing the ADC support)
- Andrew Scheller squashed several dozen bugs (usually a day or so after Dave had introduced them ;)

As always, many thanks to the whole community - we look forward to hearing from you more in 1.3!

22.7 Release 1.1.0 (2016-02-08)

- Documentation converted to reST and expanded to include generic classes and several more recipes (#80⁵¹⁹, #82⁵²⁰, #101⁵²¹, #119⁵²², #135⁵²³, #168⁵²⁴)
- New *CamJamKitRobot* (page 141) class with the pre-defined motor pins for the new CamJam EduKit
- New *LEDBarGraph* (page 116) class (many thanks to Martin O'Hanlon!) (#126⁵²⁵, #176⁵²⁶)
- New *Pin* (page 181) implementation abstracts out the concept of a GPIO pin paving the way for alternate library support and IO extenders in future (#141⁵²⁷)
- New *LEDBoard.blink()* (page 114) method which works properly even when background is set to `False` (#94⁵²⁸, #161⁵²⁹)
- New *RGBLED.blink()* (page 90) method which implements (rudimentary) color fading too! (#135⁵³⁰, #174⁵³¹)
- New `initial_value` attribute on *OutputDevice* (page 102) ensures consistent behaviour on construction (#118⁵³²)
- New `active_high` attribute on *PWMOutputDevice* (page 100) and *RGBLED* (page 90) allows use of common anode devices (#143⁵³³, #154⁵³⁴)
- Loads of new ADC chips supported (many thanks to GitHub user pcopa!) (#150⁵³⁵)

22.8 Release 1.0.0 (2015-11-16)

- Debian packaging added (#44⁵³⁶)
- *PWMLED* (page 88) class added (#58⁵³⁷)
- *TemperatureSensor* removed pending further work (#93⁵³⁸)

⁵¹⁹ <https://github.com/RPi-Distro/python-gpiozero/issues/80>
⁵²⁰ <https://github.com/RPi-Distro/python-gpiozero/issues/82>
⁵²¹ <https://github.com/RPi-Distro/python-gpiozero/issues/101>
⁵²² <https://github.com/RPi-Distro/python-gpiozero/issues/119>
⁵²³ <https://github.com/RPi-Distro/python-gpiozero/issues/135>
⁵²⁴ <https://github.com/RPi-Distro/python-gpiozero/issues/168>
⁵²⁵ <https://github.com/RPi-Distro/python-gpiozero/issues/126>
⁵²⁶ <https://github.com/RPi-Distro/python-gpiozero/issues/176>
⁵²⁷ <https://github.com/RPi-Distro/python-gpiozero/issues/141>
⁵²⁸ <https://github.com/RPi-Distro/python-gpiozero/issues/94>
⁵²⁹ <https://github.com/RPi-Distro/python-gpiozero/issues/161>
⁵³⁰ <https://github.com/RPi-Distro/python-gpiozero/issues/135>
⁵³¹ <https://github.com/RPi-Distro/python-gpiozero/issues/174>
⁵³² <https://github.com/RPi-Distro/python-gpiozero/issues/118>
⁵³³ <https://github.com/RPi-Distro/python-gpiozero/issues/143>
⁵³⁴ <https://github.com/RPi-Distro/python-gpiozero/issues/154>
⁵³⁵ <https://github.com/RPi-Distro/python-gpiozero/issues/150>
⁵³⁶ <https://github.com/RPi-Distro/python-gpiozero/issues/44>
⁵³⁷ <https://github.com/RPi-Distro/python-gpiozero/issues/58>
⁵³⁸ <https://github.com/RPi-Distro/python-gpiozero/issues/93>

- `Buzzer.beep()` (page 92) alias method added (#75⁵³⁹)
- `Motor` (page 93) PWM devices exposed, and `Robot` (page 136) motor devices exposed (#107⁵⁴⁰)

22.9 Release 0.9.0 (2015-10-25)

Fourth public beta

- Added source and values properties to all relevant classes (#76⁵⁴¹)
- Fix names of parameters in `Motor` (page 93) constructor (#79⁵⁴²)
- Added wrappers for LED groups on add-on boards (#81⁵⁴³)

22.10 Release 0.8.0 (2015-10-16)

Third public beta

- Added generic `AnalogInputDevice` (page 110) class along with specific classes for the `MCP3008` (page 107) and `MCP3004` (page 106) (#41⁵⁴⁴)
- Fixed `DigitalOutputDevice.blink()` (page 99) (#57⁵⁴⁵)

22.11 Release 0.7.0 (2015-10-09)

Second public beta

22.12 Release 0.6.0 (2015-09-28)

First public beta

22.13 Release 0.5.0 (2015-09-24)

22.14 Release 0.4.0 (2015-09-23)

22.15 Release 0.3.0 (2015-09-22)

22.16 Release 0.2.0 (2015-09-21)

Initial release

⁵³⁹ <https://github.com/RPi-Distro/python-gpiozero/issues/75>

⁵⁴⁰ <https://github.com/RPi-Distro/python-gpiozero/issues/107>

⁵⁴¹ <https://github.com/RPi-Distro/python-gpiozero/issues/76>

⁵⁴² <https://github.com/RPi-Distro/python-gpiozero/issues/79>

⁵⁴³ <https://github.com/RPi-Distro/python-gpiozero/issues/81>

⁵⁴⁴ <https://github.com/RPi-Distro/python-gpiozero/issues/41>

⁵⁴⁵ <https://github.com/RPi-Distro/python-gpiozero/issues/57>

Copyright 2015-2017 Raspberry Pi Foundation⁵⁴⁶.

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.

⁵⁴⁶ <https://www.raspberrypi.org/>

g

- `gpiozero`, 3
- `gpiozero.boards`, 113
- `gpiozero.devices`, 159
- `gpiozero.input_devices`, 73
- `gpiozero.other_devices`, 155
- `gpiozero.output_devices`, 87
- `gpiozero.pins`, 177
 - `gpiozero.pins.local`, 187
 - `gpiozero.pins.mock`, 190
 - `gpiozero.pins.native`, 189
 - `gpiozero.pins.pi`, 186
 - `gpiozero.pins.pigpio`, 188
 - `gpiozero.pins.rpigpio`, 187
 - `gpiozero.pins.rpio`, 188
- `gpiozero.spi_devices`, 105
- `gpiozero.tools`, 165

Symbols

-c, -color
pinout command line option, 60

-h, -help
pinout command line option, 60

-m, -monochrome
pinout command line option, 60

-r REVISION, -revision REVISION
pinout command line option, 60

_shared_key() (gpiozero.SharedMixin class method), 162

A

absoluted() (in module gpiozero.tools), 165

active_high (gpiozero.OutputDevice attribute), 103

active_time (gpiozero.ButtonBoard attribute), 118

active_time (gpiozero.EventsMixin attribute), 163

all_values() (in module gpiozero.tools), 168

alternating_values() (in module gpiozero.tools), 170

AnalogInputDevice (class in gpiozero), 110

angle (gpiozero.AngularServo attribute), 98

AngularServo (class in gpiozero), 96

any_values() (in module gpiozero.tools), 169

averaged() (in module gpiozero.tools), 169

B

backward() (gpiozero.CamJamKitRobot method), 141

backward() (gpiozero.Motor method), 93

backward() (gpiozero.PhaseEnableMotor method), 94

backward() (gpiozero.PhaseEnableRobot method), 138

backward() (gpiozero.PololuDRV8835Robot method), 143

backward() (gpiozero.Robot method), 137

backward() (gpiozero.RyanteckRobot method), 140

BadEventHandler, 191

BadQueueLen, 192

BadWaitTime, 192

beep() (gpiozero.Buzzer method), 92

bits (gpiozero.AnalogInputDevice attribute), 111

bits_per_word (gpiozero.SPI attribute), 184

blink() (gpiozero.DigitalOutputDevice method), 99

blink() (gpiozero.LED method), 88

blink() (gpiozero.LEDBoard method), 114

blink() (gpiozero.LedBorg method), 123

blink() (gpiozero.PiLiter method), 126

blink() (gpiozero.PiStop method), 132

blink() (gpiozero.PiTraffic method), 129

blink() (gpiozero.PWMLED method), 89

blink() (gpiozero.PWMOutputDevice method), 100

blink() (gpiozero.RGBLED method), 90

blink() (gpiozero.SnowPi method), 149

blink() (gpiozero.StatusZero method), 146

blink() (gpiozero.TrafficLights method), 121

bluetooth (gpiozero.PiBoardInfo attribute), 175

board (gpiozero.PiBoardInfo attribute), 175

booleanized() (in module gpiozero.tools), 165

bounce (gpiozero.Pin attribute), 182

Button (class in gpiozero), 73

ButtonBoard (class in gpiozero), 117

Buzzer (class in gpiozero), 92

C

CamJamKitRobot (class in gpiozero), 141

channel (gpiozero.MCP3002 attribute), 106

channel (gpiozero.MCP3004 attribute), 106

channel (gpiozero.MCP3008 attribute), 107

channel (gpiozero.MCP3202 attribute), 107

channel (gpiozero.MCP3204 attribute), 107

channel (gpiozero.MCP3208 attribute), 108

channel (gpiozero.MCP3302 attribute), 108

channel (gpiozero.MCP3304 attribute), 109

clamped() (in module gpiozero.tools), 166

clock_mode (gpiozero.SPI attribute), 184

clock_phase (gpiozero.SPI attribute), 184

clock_polarity (gpiozero.SPI attribute), 185

close() (gpiozero.CompositeDevice method), 153

close() (gpiozero.Device method), 161

close() (gpiozero.DigitalOutputDevice method), 99

close() (gpiozero.Energenie method), 144

close() (gpiozero.Factory method), 180

close() (gpiozero.GPIODevice method), 84

close() (gpiozero.LEDBoard method), 114

close() (gpiozero.LedBorg method), 123

close() (gpiozero.PiLiter method), 126

close() (gpiozero.Pin method), 181

close() (gpiozero.pins.mock.MockPin method), 190

close() (gpiozero.pins.pi.PiFactory method), 186

close() (gpiozero.PiStop method), 132
 close() (gpiozero.PiTraffic method), 130
 close() (gpiozero.PWMOutputDevice method), 101
 close() (gpiozero.SmoothedInputDevice method), 83
 close() (gpiozero.SnowPi method), 150
 close() (gpiozero.SPIDevice method), 111
 close() (gpiozero.StatusZero method), 146
 close() (gpiozero.TrafficLights method), 121
 closed (gpiozero.AngularServo attribute), 98
 closed (gpiozero.ButtonBoard attribute), 118
 closed (gpiozero.CamJamKitRobot attribute), 142
 closed (gpiozero.CompositeDevice attribute), 153
 closed (gpiozero.Device attribute), 161
 closed (gpiozero.Energenie attribute), 145
 closed (gpiozero.FishDish attribute), 135
 closed (gpiozero.GPIODevice attribute), 85
 closed (gpiozero.LEDBarGraph attribute), 117
 closed (gpiozero.LEDBoard attribute), 115
 closed (gpiozero.LedBorg attribute), 125
 closed (gpiozero.PhaseEnableRobot attribute), 139
 closed (gpiozero.PiLiter attribute), 127
 closed (gpiozero.PiLiterBarGraph attribute), 128
 closed (gpiozero.PiStop attribute), 133
 closed (gpiozero.PiTraffic attribute), 131
 closed (gpiozero.PololuDRV8835Robot attribute), 143
 closed (gpiozero.Robot attribute), 138
 closed (gpiozero.RyanteckRobot attribute), 141
 closed (gpiozero.Servo attribute), 96
 closed (gpiozero.SnowPi attribute), 151
 closed (gpiozero.SPIDevice attribute), 112
 closed (gpiozero.StatusBoard attribute), 148
 closed (gpiozero.StatusZero attribute), 147
 closed (gpiozero.TrafficHat attribute), 136
 closed (gpiozero.TrafficLights attribute), 122
 closed (gpiozero.TrafficLightsBuzzer attribute), 134
 col (gpiozero.PinInfo attribute), 176
 color (gpiozero.LedBorg attribute), 125
 color (gpiozero.RGBLED attribute), 91
 columns (gpiozero.HeaderInfo attribute), 176
 CompositeDevice (class in gpiozero), 153
 CompositeDeviceBadName, 192
 CompositeDeviceError, 192
 CompositeOutputDevice (class in gpiozero), 152
 cos_values() (in module gpiozero.tools), 170
 CPUTemperature (class in gpiozero), 156
 csi (gpiozero.PiBoardInfo attribute), 175

D

detach() (gpiozero.AngularServo method), 97
 detach() (gpiozero.Servo method), 95
 Device (class in gpiozero), 161
 DeviceClosed, 191
 differential (gpiozero.MCP3002 attribute), 106
 differential (gpiozero.MCP3004 attribute), 106
 differential (gpiozero.MCP3008 attribute), 107
 differential (gpiozero.MCP3202 attribute), 107
 differential (gpiozero.MCP3204 attribute), 108
 differential (gpiozero.MCP3208 attribute), 108

differential (gpiozero.MCP3302 attribute), 108
 differential (gpiozero.MCP3304 attribute), 109
 DigitalInputDevice (class in gpiozero), 82
 DigitalOutputDevice (class in gpiozero), 99
 distance (gpiozero.DistanceSensor attribute), 80
 DistanceSensor (class in gpiozero), 79
 dsi (gpiozero.PiBoardInfo attribute), 175

E

echo (gpiozero.DistanceSensor attribute), 81
 edges (gpiozero.Pin attribute), 182
 Energenie (class in gpiozero), 144
 EnergenieBadSocket, 192
 EnergenieSocketMissing, 192
 environment variable
 PIGPIO_ADDR, 179
 ethernet (gpiozero.PiBoardInfo attribute), 175
 EventsMixin (class in gpiozero), 162

F

Factory (class in gpiozero), 180
 FishDish (class in gpiozero), 134
 forward() (gpiozero.CamJamKitRobot method), 142
 forward() (gpiozero.Motor method), 94
 forward() (gpiozero.PhaseEnableMotor method), 94
 forward() (gpiozero.PhaseEnableRobot method), 139
 forward() (gpiozero.PololuDRV8835Robot method), 143
 forward() (gpiozero.Robot method), 137
 forward() (gpiozero.RyanteckRobot method), 140
 frame_width (gpiozero.AngularServo attribute), 98
 frame_width (gpiozero.Servo attribute), 96
 frequency (gpiozero.Pin attribute), 182
 frequency (gpiozero.PWMOutputDevice attribute), 102
 function (gpiozero.Pin attribute), 182
 function (gpiozero.PinInfo attribute), 176

G

GPIODevice (class in gpiozero), 84
 GPIODeviceClosed, 192
 GPIODeviceError, 192
 GPIOPinInUse, 192
 GPIOPinMissing, 192
 gpiozero (module), 3
 gpiozero.boards (module), 113
 gpiozero.devices (module), 159
 gpiozero.input_devices (module), 73
 gpiozero.other_devices (module), 155
 gpiozero.output_devices (module), 87
 gpiozero.pins (module), 177
 gpiozero.pins.local (module), 187
 gpiozero.pins.mock (module), 190
 gpiozero.pins.native (module), 189
 gpiozero.pins.pi (module), 186
 gpiozero.pins.pigpio (module), 188
 gpiozero.pins.rpigpio (module), 187
 gpiozero.pins.rpio (module), 188
 gpiozero.spi_devices (module), 105

gpiozero.tools (module), 165

GPIOZeroError, 191

GPIOZeroWarning, 193

H

HeaderInfo (class in gpiozero), 175

headers (gpiozero.PiBoardInfo attribute), 175

held_time (gpiozero.Button attribute), 74

held_time (gpiozero.ButtonBoard attribute), 119

held_time (gpiozero.HoldMixin attribute), 163

hold_repeat (gpiozero.Button attribute), 74

hold_repeat (gpiozero.ButtonBoard attribute), 119

hold_repeat (gpiozero.HoldMixin attribute), 163

hold_time (gpiozero.Button attribute), 74

hold_time (gpiozero.ButtonBoard attribute), 119

hold_time (gpiozero.HoldMixin attribute), 163

HoldMixin (class in gpiozero), 163

I

inactive_time (gpiozero.ButtonBoard attribute), 119

inactive_time (gpiozero.EventsMixin attribute), 163

input_with_pull() (gpiozero.Pin method), 181

InputDevice (class in gpiozero), 84

InputDeviceError, 192

InternalDevice (class in gpiozero), 157

inverted() (in module gpiozero.tools), 166

is_active (gpiozero.AngularServo attribute), 98

is_active (gpiozero.ButtonBoard attribute), 119

is_active (gpiozero.Buzzer attribute), 93

is_active (gpiozero.CamJamKitRobot attribute), 142

is_active (gpiozero.CompositeDevice attribute), 153

is_active (gpiozero.CPUTemperature attribute), 157

is_active (gpiozero.Device attribute), 161

is_active (gpiozero.Energenie attribute), 145

is_active (gpiozero.FishDish attribute), 135

is_active (gpiozero.LEDBarGraph attribute), 117

is_active (gpiozero.LEDBoard attribute), 115

is_active (gpiozero.LedBorg attribute), 125

is_active (gpiozero.PhaseEnableRobot attribute), 139

is_active (gpiozero.PiLiter attribute), 127

is_active (gpiozero.PiLiterBarGraph attribute), 128

is_active (gpiozero.PiStop attribute), 133

is_active (gpiozero.PiTrafic attribute), 131

is_active (gpiozero.PololuDRV8835Robot attribute), 144

is_active (gpiozero.PWMOutputDevice attribute), 102

is_active (gpiozero.Robot attribute), 138

is_active (gpiozero.RyanteckRobot attribute), 141

is_active (gpiozero.Servo attribute), 96

is_active (gpiozero.SmoothedInputDevice attribute), 84

is_active (gpiozero.SnowPi attribute), 151

is_active (gpiozero.StatusBoard attribute), 148

is_active (gpiozero.StatusZero attribute), 147

is_active (gpiozero.TrafficHat attribute), 136

is_active (gpiozero.TrafficLights attribute), 122

is_active (gpiozero.TrafficLightsBuzzer attribute), 134

is_held (gpiozero.Button attribute), 74

is_held (gpiozero.ButtonBoard attribute), 119

is_held (gpiozero.HoldMixin attribute), 163

is_lit (gpiozero.LED attribute), 88

is_lit (gpiozero.LedBorg attribute), 125

is_lit (gpiozero.PWMLed attribute), 89

is_lit (gpiozero.RGBLED attribute), 92

is_pressed (gpiozero.Button attribute), 74

is_pressed (gpiozero.ButtonBoard attribute), 119

L

LED (class in gpiozero), 87

LEDBarGraph (class in gpiozero), 116

LEDBoard (class in gpiozero), 113

LedBorg (class in gpiozero), 123

LEDCollection (class in gpiozero), 152

leds (gpiozero.LEDBarGraph attribute), 117

leds (gpiozero.LEDBoard attribute), 115

leds (gpiozero.LEDCollection attribute), 152

leds (gpiozero.PiLiter attribute), 127

leds (gpiozero.PiLiterBarGraph attribute), 128

leds (gpiozero.PiStop attribute), 133

leds (gpiozero.PiTrafic attribute), 131

leds (gpiozero.SnowPi attribute), 151

leds (gpiozero.StatusZero attribute), 147

leds (gpiozero.TrafficLights attribute), 122

left() (gpiozero.CamJamKitRobot method), 142

left() (gpiozero.PhaseEnableRobot method), 139

left() (gpiozero.PololuDRV8835Robot method), 143

left() (gpiozero.Robot method), 137

left() (gpiozero.RyanteckRobot method), 140

light_detected (gpiozero.LightSensor attribute), 79

LightSensor (class in gpiozero), 78

LineSensor (class in gpiozero), 75

lit_count (gpiozero.LEDBarGraph attribute), 117

lit_count (gpiozero.PiLiterBarGraph attribute), 128

LocalPiFactory (class in gpiozero.pins.local), 187

LocalPiPin (class in gpiozero.pins.local), 187

lsb_first (gpiozero.SPI attribute), 185

M

manufacturer (gpiozero.PiBoardInfo attribute), 174

max() (gpiozero.AngularServo method), 97

max() (gpiozero.Servo method), 95

max_angle (gpiozero.AngularServo attribute), 98

max_distance (gpiozero.DistanceSensor attribute), 81

max_pulse_width (gpiozero.AngularServo attribute), 98

max_pulse_width (gpiozero.Servo attribute), 96

max_voltage (gpiozero.AnalogInputDevice attribute), 111

MCP3001 (class in gpiozero), 106

MCP3002 (class in gpiozero), 106

MCP3004 (class in gpiozero), 106

MCP3008 (class in gpiozero), 107

MCP3201 (class in gpiozero), 107

MCP3202 (class in gpiozero), 107

MCP3204 (class in gpiozero), 107

MCP3208 (class in gpiozero), 108

MCP3301 (class in gpiozero), 108

MCP3302 (class in gpiozero), 108
MCP3304 (class in gpiozero), 109
memory (gpiozero.PiBoardInfo attribute), 174
mid() (gpiozero.AngularServo method), 97
mid() (gpiozero.Servo method), 96
min() (gpiozero.AngularServo method), 98
min() (gpiozero.Servo method), 96
min_angle (gpiozero.AngularServo attribute), 98
min_pulse_width (gpiozero.AngularServo attribute), 98
min_pulse_width (gpiozero.Servo attribute), 96
MockChargingPin (class in gpiozero.pins.mock), 190
MockConnectedPin (class in gpiozero.pins.mock), 190
MockFactory (class in gpiozero.pins.mock), 190
MockPin (class in gpiozero.pins.mock), 190
MockPWMPin (class in gpiozero.pins.mock), 190
MockTriggerPin (class in gpiozero.pins.mock), 190
model (gpiozero.PiBoardInfo attribute), 174
motion_detected (gpiozero.MotionSensor attribute), 77
MotionSensor (class in gpiozero), 76
Motor (class in gpiozero), 93
multiplied() (in module gpiozero.tools), 169

N

name (gpiozero.HeaderInfo attribute), 176
NativeFactory (class in gpiozero.pins.native), 189
NativePin (class in gpiozero.pins.native), 189
negated() (in module gpiozero.tools), 166
number (gpiozero.PinInfo attribute), 176

O

off() (gpiozero.Buzzer method), 92
off() (gpiozero.CompositeOutputDevice method), 152
off() (gpiozero.DigitalOutputDevice method), 100
off() (gpiozero.FishDish method), 135
off() (gpiozero.LED method), 88
off() (gpiozero.LEDBarGraph method), 117
off() (gpiozero.LEDBoard method), 115
off() (gpiozero.LedBorg method), 124
off() (gpiozero.OutputDevice method), 102
off() (gpiozero.PiLiter method), 127
off() (gpiozero.PiLiterBarGraph method), 128
off() (gpiozero.PiStop method), 133
off() (gpiozero.PiTraffic method), 130
off() (gpiozero.PWMLED method), 89
off() (gpiozero.PWMOutputDevice method), 101
off() (gpiozero.RGBLED method), 91
off() (gpiozero.SnowPi method), 150
off() (gpiozero.StatusBoard method), 148
off() (gpiozero.StatusZero method), 147
off() (gpiozero.TrafficHat method), 136
off() (gpiozero.TrafficLights method), 122
off() (gpiozero.TrafficLightsBuzzer method), 134
on() (gpiozero.Buzzer method), 93
on() (gpiozero.CompositeOutputDevice method), 152
on() (gpiozero.DigitalOutputDevice method), 100
on() (gpiozero.FishDish method), 135
on() (gpiozero.LED method), 88
on() (gpiozero.LEDBarGraph method), 117

on() (gpiozero.LEDBoard method), 115
on() (gpiozero.LedBorg method), 124
on() (gpiozero.OutputDevice method), 102
on() (gpiozero.PiLiter method), 127
on() (gpiozero.PiLiterBarGraph method), 128
on() (gpiozero.PiStop method), 133
on() (gpiozero.PiTraffic method), 130
on() (gpiozero.PWMLED method), 89
on() (gpiozero.PWMOutputDevice method), 101
on() (gpiozero.RGBLED method), 91
on() (gpiozero.SnowPi method), 150
on() (gpiozero.StatusBoard method), 148
on() (gpiozero.StatusZero method), 147
on() (gpiozero.TrafficHat method), 136
on() (gpiozero.TrafficLights method), 122
on() (gpiozero.TrafficLightsBuzzer method), 134
output_with_state() (gpiozero.Pin method), 182
OutputDevice (class in gpiozero), 102
OutputDeviceBadValue, 192
OutputDeviceError, 192

P

partial (gpiozero.SmoothedInputDevice attribute), 84
pcb_revision (gpiozero.PiBoardInfo attribute), 174
PhaseEnableMotor (class in gpiozero), 94
PhaseEnableRobot (class in gpiozero), 138
physical_pin() (gpiozero.PiBoardInfo method), 173
physical_pins() (gpiozero.PiBoardInfo method), 174
pi_info (gpiozero.Factory attribute), 181
pi_info() (in module gpiozero), 173
PiBoardInfo (class in gpiozero), 173
PiFactory (class in gpiozero.pins.pi), 186
PIGPIO_ADDR, 179
PiGPIOFactory (class in gpiozero.pins.pigpio), 188
PiGIOPin (class in gpiozero.pins.pigpio), 189
PiLiter (class in gpiozero), 125
PiLiterBarGraph (class in gpiozero), 128
Pin (class in gpiozero), 181
pin (gpiozero.Button attribute), 74
pin (gpiozero.Buzzer attribute), 93
pin (gpiozero.GPIODevice attribute), 85
pin (gpiozero.LED attribute), 88
pin (gpiozero.LightSensor attribute), 79
pin (gpiozero.LineSensor attribute), 76
pin (gpiozero.MotionSensor attribute), 77
pin (gpiozero.PWMLED attribute), 90
pin() (gpiozero.Factory method), 180
pin() (gpiozero.pins.mock.MockFactory method), 190
pin() (gpiozero.pins.pi.PiFactory method), 186
PinEdgeDetectUnsupported, 193
PinError, 192
PinFactoryFallback, 193
PinFixedPull, 193
PingServer (class in gpiozero), 156
PinInfo (class in gpiozero), 176
PinInvalidBounce, 193
PinInvalidEdges, 193
PinInvalidFunction, 193

[PinInvalidPin](#), 193
[PinInvalidPull](#), 193
[PinInvalidState](#), 193
[PinMultiplePins](#), 193
[PinNonPhysical](#), 194
[PinNoPins](#), 193
[pinout](#) command line option
 -c, --color, 60
 -h, --help, 60
 -m, --monochrome, 60
 -r REVISION, --revision REVISION, 60
[PinPWMError](#), 193
[PinPWMFxedValue](#), 193
[PinPWMUnsupported](#), 193
[pins](#) (gpiozero.HeaderInfo attribute), 176
[PinSetInput](#), 193
[PinSPIUnsupported](#), 193
[PinUnknownPi](#), 193
[PinUnsupported](#), 193
[PiPin](#) (class in gpiozero.pins.pi), 187
[PiStop](#) (class in gpiozero), 131
[PiTraffic](#) (class in gpiozero), 129
[PololuDRV8835Robot](#) (class in gpiozero), 143
[post_delayed\(\)](#) (in module gpiozero.tools), 166
[post_periodic_filtered\(\)](#) (in module gpiozero.tools), 166
[pprint\(\)](#) (gpiozero.HeaderInfo method), 176
[pprint\(\)](#) (gpiozero.PiBoardInfo method), 174
[pre_delayed\(\)](#) (in module gpiozero.tools), 167
[pre_periodic_filtered\(\)](#) (in module gpiozero.tools), 167
[pressed_time](#) (gpiozero.ButtonBoard attribute), 119
[pull](#) (gpiozero.Pin attribute), 183
[pull_up](#) (gpiozero.Button attribute), 74
[pull_up](#) (gpiozero.ButtonBoard attribute), 119
[pull_up](#) (gpiozero.InputDevice attribute), 84
[pull_up](#) (gpiozero.PinInfo attribute), 176
[pulled_up\(\)](#) (gpiozero.PiBoardInfo method), 174
[pulse\(\)](#) (gpiozero.LEDBoard method), 115
[pulse\(\)](#) (gpiozero.LedBorg method), 124
[pulse\(\)](#) (gpiozero.PiLiter method), 127
[pulse\(\)](#) (gpiozero.PiStop method), 133
[pulse\(\)](#) (gpiozero.PiTraffic method), 130
[pulse\(\)](#) (gpiozero.PWMLED method), 89
[pulse\(\)](#) (gpiozero.PWMOutputDevice method), 101
[pulse\(\)](#) (gpiozero.RGBLED method), 91
[pulse\(\)](#) (gpiozero.SnowPi method), 150
[pulse\(\)](#) (gpiozero.StatusZero method), 147
[pulse\(\)](#) (gpiozero.TrafficLights method), 122
[pulse_width](#) (gpiozero.AngularServo attribute), 98
[pulse_width](#) (gpiozero.Servo attribute), 96
[PWMLED](#) (class in gpiozero), 88
[PWMOutputDevice](#) (class in gpiozero), 100

Q

[quantized\(\)](#) (in module gpiozero.tools), 167
[queue_len](#) (gpiozero.SmoothedInputDevice attribute), 84
[queued\(\)](#) (in module gpiozero.tools), 167

R

[ramping_values\(\)](#) (in module gpiozero.tools), 170
[random_values\(\)](#) (in module gpiozero.tools), 171
[raw_value](#) (gpiozero.AnalogInputDevice attribute), 111
[read\(\)](#) (gpiozero.SPI method), 184
[release_all\(\)](#) (gpiozero.Factory method), 180
[release_pins\(\)](#) (gpiozero.Factory method), 180
[released](#) (gpiozero.PiBoardInfo attribute), 174
[reserve_pins\(\)](#) (gpiozero.Factory method), 180
[reset\(\)](#) (gpiozero.pins.mock.MockFactory method), 190
[reverse\(\)](#) (gpiozero.CamJamKitRobot method), 142
[reverse\(\)](#) (gpiozero.Motor method), 94
[reverse\(\)](#) (gpiozero.PhaseEnableMotor method), 94
[reverse\(\)](#) (gpiozero.PhaseEnableRobot method), 139
[reverse\(\)](#) (gpiozero.PololuDRV8835Robot method), 143
[reverse\(\)](#) (gpiozero.Robot method), 137
[reverse\(\)](#) (gpiozero.RyanteckRobot method), 140
[revision](#) (gpiozero.PiBoardInfo attribute), 174
[RGBLED](#) (class in gpiozero), 90
[right\(\)](#) (gpiozero.CamJamKitRobot method), 142
[right\(\)](#) (gpiozero.PhaseEnableRobot method), 139
[right\(\)](#) (gpiozero.PololuDRV8835Robot method), 143
[right\(\)](#) (gpiozero.Robot method), 137
[right\(\)](#) (gpiozero.RyanteckRobot method), 140
[Robot](#) (class in gpiozero), 136
[row](#) (gpiozero.PinInfo attribute), 176
[rows](#) (gpiozero.HeaderInfo attribute), 176
[RPiGPIOFactory](#) (class in gpiozero.pins.rpigpio), 187
[RPiGPIOPin](#) (class in gpiozero.pins.rpigpio), 188
[RPIOFactory](#) (class in gpiozero.pins.rpio), 188
[RPIOPin](#) (class in gpiozero.pins.rpio), 188
[RyanteckRobot](#) (class in gpiozero), 139

S

[scaled\(\)](#) (in module gpiozero.tools), 168
[select_high](#) (gpiozero.SPI attribute), 186
[Servo](#) (class in gpiozero), 95
[SharedMixin](#) (class in gpiozero), 162
[sin_values\(\)](#) (in module gpiozero.tools), 171
[smoothed\(\)](#) (in module gpiozero.tools), 168
[SmoothedInputDevice](#) (class in gpiozero), 82
[SnowPi](#) (class in gpiozero), 149
[soc](#) (gpiozero.PiBoardInfo attribute), 174
[source](#) (gpiozero.AngularServo attribute), 98
[source](#) (gpiozero.CamJamKitRobot attribute), 142
[source](#) (gpiozero.Energenie attribute), 145
[source](#) (gpiozero.FishDish attribute), 135
[source](#) (gpiozero.LEDBarGraph attribute), 117
[source](#) (gpiozero.LEDBoard attribute), 115
[source](#) (gpiozero.LedBorg attribute), 125
[source](#) (gpiozero.PhaseEnableRobot attribute), 139
[source](#) (gpiozero.PiLiter attribute), 127
[source](#) (gpiozero.PiLiterBarGraph attribute), 128
[source](#) (gpiozero.PiStop attribute), 133
[source](#) (gpiozero.PiTraffic attribute), 131
[source](#) (gpiozero.PololuDRV8835Robot attribute), 144
[source](#) (gpiozero.Robot attribute), 138

source (gpiozero.RyanteckRobot attribute), 141
source (gpiozero.Servo attribute), 96
source (gpiozero.SnowPi attribute), 151
source (gpiozero.SourceMixin attribute), 162
source (gpiozero.StatusBoard attribute), 148
source (gpiozero.StatusZero attribute), 147
source (gpiozero.TrafficHat attribute), 136
source (gpiozero.TrafficLights attribute), 122
source (gpiozero.TrafficLightsBuzzer attribute), 134
source_delay (gpiozero.AngularServo attribute), 98
source_delay (gpiozero.CamJamKitRobot attribute), 142
source_delay (gpiozero.Energenie attribute), 145
source_delay (gpiozero.FishDish attribute), 135
source_delay (gpiozero.LEDBarGraph attribute), 117
source_delay (gpiozero.LEDBoard attribute), 115
source_delay (gpiozero.LedBorg attribute), 125
source_delay (gpiozero.PhaseEnableRobot attribute), 139
source_delay (gpiozero.PiLiter attribute), 127
source_delay (gpiozero.PiLiterBarGraph attribute), 128
source_delay (gpiozero.PiStop attribute), 133
source_delay (gpiozero.PiTraffic attribute), 131
source_delay (gpiozero.PololuDRV8835Robot attribute), 144
source_delay (gpiozero.Robot attribute), 138
source_delay (gpiozero.RyanteckRobot attribute), 141
source_delay (gpiozero.Servo attribute), 96
source_delay (gpiozero.SnowPi attribute), 151
source_delay (gpiozero.SourceMixin attribute), 162
source_delay (gpiozero.StatusBoard attribute), 148
source_delay (gpiozero.StatusZero attribute), 147
source_delay (gpiozero.TrafficHat attribute), 136
source_delay (gpiozero.TrafficLights attribute), 122
source_delay (gpiozero.TrafficLightsBuzzer attribute), 134
SourceMixin (class in gpiozero), 162
SPI (class in gpiozero), 183
spi() (gpiozero.Factory method), 181
spi() (gpiozero.pins.pi.PiFactory method), 186
SPIBadArgs, 192
SPIBadChannel, 192
SPIDevice (class in gpiozero), 111
SPIError, 192
SPIFixedBitOrder, 192
SPIFixedClockMode, 192
SPIFixedSelect, 192
SPIFixedWordSize, 192
SPIInvalidClockMode, 192
SPIInvalidWordSize, 192
SPISoftwareFallback, 193
SPIWarning, 193
state (gpiozero.Pin attribute), 183
StatusBoard (class in gpiozero), 148
StatusZero (class in gpiozero), 145
stop() (gpiozero.CamJamKitRobot method), 142
stop() (gpiozero.Motor method), 94
stop() (gpiozero.PhaseEnableMotor method), 95

stop() (gpiozero.PhaseEnableRobot method), 139
stop() (gpiozero.PololuDRV8835Robot method), 143
stop() (gpiozero.Robot method), 138
stop() (gpiozero.RyanteckRobot method), 141
storage (gpiozero.PiBoardInfo attribute), 175
summed() (in module gpiozero.tools), 169

T

temperature (gpiozero.CPUTemperature attribute), 157
threshold (gpiozero.SmoothedInputDevice attribute), 84
threshold_distance (gpiozero.DistanceSensor attribute), 81
TimeOfDay (class in gpiozero), 155
toggle() (gpiozero.Buzzer method), 93
toggle() (gpiozero.CompositeOutputDevice method), 152
toggle() (gpiozero.FishDish method), 135
toggle() (gpiozero.LED method), 88
toggle() (gpiozero.LEDBarGraph method), 117
toggle() (gpiozero.LEDBoard method), 115
toggle() (gpiozero.LedBorg method), 125
toggle() (gpiozero.OutputDevice method), 102
toggle() (gpiozero.PiLiter method), 127
toggle() (gpiozero.PiLiterBarGraph method), 128
toggle() (gpiozero.PiStop method), 133
toggle() (gpiozero.PiTraffic method), 131
toggle() (gpiozero.PWMLED method), 89
toggle() (gpiozero.PWMOutputDevice method), 102
toggle() (gpiozero.RGBLED method), 91
toggle() (gpiozero.SnowPi method), 150
toggle() (gpiozero.StatusBoard method), 148
toggle() (gpiozero.StatusZero method), 147
toggle() (gpiozero.TrafficHat method), 136
toggle() (gpiozero.TrafficLights method), 122
toggle() (gpiozero.TrafficLightsBuzzer method), 134
TrafficHat (class in gpiozero), 135
TrafficLights (class in gpiozero), 120
TrafficLightsBuzzer (class in gpiozero), 134
transfer() (gpiozero.SPI method), 184
trigger (gpiozero.DistanceSensor attribute), 81

U

usb (gpiozero.PiBoardInfo attribute), 175

V

value (gpiozero.AnalogInputDevice attribute), 111
value (gpiozero.AngularServo attribute), 98
value (gpiozero.ButtonBoard attribute), 119
value (gpiozero.CamJamKitRobot attribute), 142
value (gpiozero.CompositeDevice attribute), 153
value (gpiozero.CompositeOutputDevice attribute), 152
value (gpiozero.Device attribute), 161
value (gpiozero.DigitalOutputDevice attribute), 100
value (gpiozero.Energenie attribute), 145
value (gpiozero.FishDish attribute), 135
value (gpiozero.GPIODevice attribute), 85
value (gpiozero.LEDBarGraph attribute), 117

value (gpiozero.LEDBoard attribute), 116
 value (gpiozero.LedBorg attribute), 125
 value (gpiozero.MCP3001 attribute), 106
 value (gpiozero.MCP3002 attribute), 106
 value (gpiozero.MCP3004 attribute), 107
 value (gpiozero.MCP3008 attribute), 107
 value (gpiozero.MCP3201 attribute), 107
 value (gpiozero.MCP3202 attribute), 107
 value (gpiozero.MCP3204 attribute), 108
 value (gpiozero.MCP3208 attribute), 108
 value (gpiozero.MCP3301 attribute), 108
 value (gpiozero.MCP3302 attribute), 109
 value (gpiozero.MCP3304 attribute), 109
 value (gpiozero.OutputDevice attribute), 103
 value (gpiozero.PhaseEnableRobot attribute), 139
 value (gpiozero.PiLiter attribute), 127
 value (gpiozero.PiLiterBarGraph attribute), 128
 value (gpiozero.PiStop attribute), 133
 value (gpiozero.PiTrafic attribute), 131
 value (gpiozero.PololuDRV8835Robot attribute), 144
 value (gpiozero.PWMLED attribute), 90
 value (gpiozero.PWMOutputDevice attribute), 102
 value (gpiozero.Robot attribute), 138
 value (gpiozero.RyanteckRobot attribute), 141
 value (gpiozero.Servo attribute), 96
 value (gpiozero.SmoothedInputDevice attribute), 84
 value (gpiozero.SnowPi attribute), 151
 value (gpiozero.StatusBoard attribute), 148
 value (gpiozero.StatusZero attribute), 148
 value (gpiozero.TrafficHat attribute), 136
 value (gpiozero.TrafficLights attribute), 122
 value (gpiozero.TrafficLightsBuzzer attribute), 134
 values (gpiozero.AngularServo attribute), 98
 values (gpiozero.ButtonBoard attribute), 119
 values (gpiozero.CamJamKitRobot attribute), 143
 values (gpiozero.Energenie attribute), 145
 values (gpiozero.FishDish attribute), 135
 values (gpiozero.LEDBarGraph attribute), 117
 values (gpiozero.LEDBoard attribute), 116
 values (gpiozero.LedBorg attribute), 125
 values (gpiozero.PhaseEnableRobot attribute), 139
 values (gpiozero.PiLiter attribute), 127
 values (gpiozero.PiLiterBarGraph attribute), 129
 values (gpiozero.PiStop attribute), 133
 values (gpiozero.PiTrafic attribute), 131
 values (gpiozero.PololuDRV8835Robot attribute), 144
 values (gpiozero.Robot attribute), 138
 values (gpiozero.RyanteckRobot attribute), 141
 values (gpiozero.Servo attribute), 96
 values (gpiozero.SnowPi attribute), 151
 values (gpiozero.StatusBoard attribute), 149
 values (gpiozero.StatusZero attribute), 148
 values (gpiozero.TrafficHat attribute), 136
 values (gpiozero.TrafficLights attribute), 122
 values (gpiozero.TrafficLightsBuzzer attribute), 134
 values (gpiozero.ValuesMixin attribute), 162
 ValuesMixin (class in gpiozero), 161
 voltage (gpiozero.AnalogInputDevice attribute), 111

W

wait_for_active() (gpiozero.ButtonBoard method), 118
 wait_for_active() (gpiozero.EventsMixin method), 162
 wait_for_dark() (gpiozero.LightSensor method), 79
 wait_for_in_range() (gpiozero.DistanceSensor method), 80
 wait_for_inactive() (gpiozero.ButtonBoard method), 118
 wait_for_inactive() (gpiozero.EventsMixin method), 163
 wait_for_light() (gpiozero.LightSensor method), 79
 wait_for_line() (gpiozero.LineSensor method), 76
 wait_for_motion() (gpiozero.MotionSensor method), 77
 wait_for_no_line() (gpiozero.LineSensor method), 76
 wait_for_no_motion() (gpiozero.MotionSensor method), 77
 wait_for_out_of_range() (gpiozero.DistanceSensor method), 80
 wait_for_press() (gpiozero.Button method), 74
 wait_for_press() (gpiozero.ButtonBoard method), 118
 wait_for_release() (gpiozero.Button method), 74
 wait_for_release() (gpiozero.ButtonBoard method), 118
 when_activated (gpiozero.ButtonBoard attribute), 119
 when_activated (gpiozero.EventsMixin attribute), 163
 when_changed (gpiozero.Pin attribute), 183
 when_dark (gpiozero.LightSensor attribute), 79
 when_deactivated (gpiozero.ButtonBoard attribute), 119
 when_deactivated (gpiozero.EventsMixin attribute), 163
 when_held (gpiozero.Button attribute), 74
 when_held (gpiozero.ButtonBoard attribute), 119
 when_held (gpiozero.HoldMixin attribute), 163
 when_in_range (gpiozero.DistanceSensor attribute), 81
 when_light (gpiozero.LightSensor attribute), 79
 when_line (gpiozero.LineSensor attribute), 76
 when_motion (gpiozero.MotionSensor attribute), 77
 when_no_line (gpiozero.LineSensor attribute), 76
 when_no_motion (gpiozero.MotionSensor attribute), 78
 when_out_of_range (gpiozero.DistanceSensor attribute), 81
 when_pressed (gpiozero.Button attribute), 75
 when_pressed (gpiozero.ButtonBoard attribute), 120
 when_released (gpiozero.Button attribute), 75
 when_released (gpiozero.ButtonBoard attribute), 120
 wifi (gpiozero.PiBoardInfo attribute), 175
 write() (gpiozero.SPI method), 184