<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Installing GPIO Zero</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>Basic Recipes</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>Advanced Recipes</td>
<td>35</td>
</tr>
<tr>
<td>4</td>
<td>Configuring Remote GPIO</td>
<td>43</td>
</tr>
<tr>
<td>5</td>
<td>Remote GPIO Recipes</td>
<td>51</td>
</tr>
<tr>
<td>6</td>
<td>Pi Zero USB OTG</td>
<td>55</td>
</tr>
<tr>
<td>7</td>
<td>Source/Values</td>
<td>59</td>
</tr>
<tr>
<td>8</td>
<td>Command-line Tools</td>
<td>69</td>
</tr>
<tr>
<td>9</td>
<td>Frequently Asked Questions</td>
<td>75</td>
</tr>
<tr>
<td>10</td>
<td>Migrating from RPi.GPIO</td>
<td>81</td>
</tr>
<tr>
<td>11</td>
<td>Contributing</td>
<td>87</td>
</tr>
<tr>
<td>12</td>
<td>Development</td>
<td>89</td>
</tr>
<tr>
<td>13</td>
<td>API - Input Devices</td>
<td>93</td>
</tr>
<tr>
<td>14</td>
<td>API - Output Devices</td>
<td>111</td>
</tr>
<tr>
<td>15</td>
<td>API - SPI Devices</td>
<td>133</td>
</tr>
<tr>
<td>16</td>
<td>API - Boards and Accessories</td>
<td>141</td>
</tr>
<tr>
<td>17</td>
<td>API - Internal Devices</td>
<td>167</td>
</tr>
<tr>
<td>18</td>
<td>API - Generic Classes</td>
<td>173</td>
</tr>
<tr>
<td>19</td>
<td>API - Device Source Tools</td>
<td>179</td>
</tr>
<tr>
<td>20</td>
<td>API - Tones</td>
<td>187</td>
</tr>
<tr>
<td>21</td>
<td>API - Pi Information</td>
<td>189</td>
</tr>
<tr>
<td>22</td>
<td>API - Pins</td>
<td>195</td>
</tr>
</tbody>
</table>
Installing GPIO Zero

GPIO Zero is installed by default in the Raspbian\(^1\) image, and the Raspberry Pi Desktop\(^2\) image for PC/Mac, both available from raspberrypi.org\(^3\). Follow these guides to installing on Raspbian Lite and other operating systems, including for PCs using the remote GPIO (page 43) 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\(^4\) 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 89) page.

---

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 43) 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` interface from GPIO Zero, it should be explicitly imported:

```python
from gpiozero import Button
```

Now `Button` is available directly in your script:

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

Alternatively, the whole GPIO Zero library can be imported:

```python
import gpiozero
```

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

```python
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` library, this is not configurable. However, translation from other schemes can be used by providing prefixes to pin numbers (see below).

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
If you wish to use physical (BOARD) numbering you can specify the pin number as "BOARD11". If you are familiar with the wiringPi\textsuperscript{6} pin numbers (another physical layout) you could use “WPI0” instead. Finally, you can specify pins as “header:number”, e.g. “J8:11” meaning physical pin 11 on header J8 (the GPIO header on modern Pis). Hence, the following lines are all equivalent:

```python
>>> led = LED(17)
```

\textsuperscript{6} https://projects.drogon.net/raspberry-pi/wiringpi/pins/
Note that these alternate schemes are merely translations. If you request the state of a device on the command line, the associated pin number will always be reported in the Broadcom (BCM) scheme:

```python
>>> led = LED("BOARD11")
>>> led
<gpiozero.LED object on pin GPIO17, active_high=True, is_active=False>
```

Throughout this manual we will use the default integer pin numbers, in the Broadcom (BCM) layout shown above.

### 2.3 LED

Turn an LED (page 111) on and off repeatedly:

```python
from gpiozero import LED
from time import sleep

red = LED(17)

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

Alternatively:

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

(continues on next page)
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()` 7. See How do I keep my script running? (page 75) for more information.

2.4 LED with variable brightness

![LED Circuit Diagram]

Any regular LED can have its brightness value set using PWM (pulse-width-modulation). In GPIO Zero, this can be achieved using `PWMLED` (page 113) using values between 0 and 1:

```python
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):

---

7 https://docs.python.org/3.5/library/signal.html#signal.pause
from gpiozero import PWMLED
from signal import pause

led = PWMLED(17)
led.pulse()
pause()

2.5 Button

Check if a Button (page 93) 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

(continues on next page)
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()

8 https://docs.python.org/3.5/library/constants.html#None
2.6 Button controlled LED

Turn on an **LED** (page 111) when a **Button** (page 93) is pressed:

```python
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:

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

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

led.source = button

pause()
```

2.7 Button controlled camera

Using the button press to trigger **PiCamera**\(^9\) to take a picture using `button.when_pressed = camera.capture()` would not work because the `camera.capture()` method requires an `output` parameter. However, this can be achieved using a custom function which requires no parameters:


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

button = Button(2)
camera = PiCamera()

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

button.when_pressed = capture

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

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

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

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

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

```python
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():
    timestamp = datetime.now().isoformat()
    camera.capture('/home/pi/%s' % timestamp)

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

```

### 2.8 Shutdown button

The `Button` (page 93) 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:

```python
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

```
2.9 LEDBoard

A collection of LEDs can be accessed using `LEDBoard` (page 141):

```python
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 141) with `pwm=True` allows each LED’s brightness to be controlled:

```python
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 141) examples in the advanced `LEDBoard recipes` (page 35).
A collection of LEDs can be treated like a bar graph using LEDBarGraph (page 144):

```python
from gpiozero import LEDBarGraph
from time import sleep
from __future__ import division  # required for python 2

graph = LEDBarGraph(5, 6, 13, 19, 26, 20)

graph.value = 1  # (1, 1, 1, 1, 1, 1)
sleep(1)

graph.value = 1/2  # (1, 1, 1, 0, 0, 0)
sleep(1)

graph.value = -1/2  # (0, 0, 0, 1, 1, 1)
sleep(1)

graph.value = 1/4  # (1, 0, 0, 0, 0, 0)
sleep(1)

graph.value = -1  # (1, 1, 1, 1, 1, 1)
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 144) with pwm=True allows more precise values using LED brightness:

```python
from gpiozero import LEDBarGraph
from time import sleep
from __future__ import division  # required for python 2

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)
```

(continues on next page)
A full traffic lights system.

Using a **TrafficLights** (page 147) kit like Pi-Stop:

```python
from gpiozero import TrafficLights
from time import sleep

lights = TrafficLights(2, 3, 4)
lights.green.on()

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

```python
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 111) components:

```python
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:

```python
from gpiozero import Button
from picamera import PiCamera

button = Button(2)
camera = PiCamera()

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


### 2.13 Reaction Game

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

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

led = LED(17)
player_1 = Button(2)
player_2 = Button(3)

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

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

led.off()

See Quick Reaction Game\textsuperscript{12} for a full resource.

### 2.14 GPIO Music Box

Each button plays a different sound!

```python
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\textsuperscript{13} for a full resource.

### 2.15 All on when pressed

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

\textsuperscript{12} https://projects.raspberrypi.org/en/projects/python-quick-reaction-game

**FishDish** (page 153):

```python
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 153):

```python
from gpiozero import TrafficHat
from signal import pause

th = TrafficHat()
th.button.when_pressed = th.on
th.button.when_released = th.off

pause()
```

Using **LED** (page 111), **Buzzer** (page 117), and **Button** (page 93) components:

```python
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 115):

```python
from gpiozero import RGBLED
from time import sleep
from __future__ import division  # required for python 2

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 111) when a MotionSensor (page 97) detects motion:

```python
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 99) detect light and dark:

```python
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:

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

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

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

pause()
```

Or make a `PWMLED` (page 113) change brightness according to the detected light level:

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

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

led.source = sensor

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 the sensor’s diagram obscuring most of the breadboard!)

Have a `DistanceSensor` (page 101) detect the distance to the nearest object:

```python
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:

```python
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 Servo

Control a servo between its minimum, mid-point and maximum positions in sequence:
from gpiozero import Servo
from time import sleep

servo = Servo(17)

while True:
    servo.min()
    sleep(2)
    servo.mid()
    sleep(2)
    servo.max()
    sleep(2)

Use a button to control the servo between its minimum and maximum positions:

from gpiozero import Servo, Button

servo = Servo(17)
btn = Button(14)

while True:
    servo.min()
    btn.wait_for_press()
    servo.max()
    btn.wait_for_press()

Automate the servo to continuously slowly sweep:

from gpiozero import Servo
from gpiozero.tools import sin_values

servo = Servo(17)

servo.source = sin_values()
servo.source_delay = 0.1

Use AngularServo (page 124) so you can specify an angle:

from gpiozero import AngularServo
from time import sleep

servo = AngularServo(17, min_angle=-90, max_angle=90)

while True:
    servo.angle = -90
    sleep(2)
    servo.angle = -45
    sleep(2)
    servo.angle = 0
    sleep(2)
    servo.angle = 45
    sleep(2)
    servo.angle = 90
    sleep(2)
2.21 Motors

Spin a Motor (page 120) around forwards and backwards:

```python
from gpiozero import Motor
from time import sleep

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

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

Make a Robot (page 155) drive around in (roughly) a square:

```python
from gpiozero import Robot
from time import sleep

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

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

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

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

(distance sensor code here)
```

(continues on next page)
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()

2.23 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()
2.24 Keyboard controlled robot

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

```python
import curses
from gpiozero import Robot

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

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

def main(window):
    next_key = None
    while True:
        curses.halfdelay(1)
        (continues on next page)```
if next_key is None:
    key = window.getch()
else:
    key = next_key
next_key = None
if key != -1:
    # KEY PRESSED
    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 RELEASED
    robot.stop()
curses.wrapper(main)

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

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

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

robot = Robot(left=(4, 14), right=(17, 18))
# Get the list of available input devices
devices = [InputDevice(device) for device in list_devices()]
# Filter out everything that's not a keyboard. Keyboards are defined as any
# device which has keys, and which specifically has keys 1..31 (roughly Esc,
# the numeric keys, the first row of QWERTY plus a few more) and which does
# *not* have key 0 (reserved)
must_have = {i for i in range(1, 32)}
must_not_have = {0}
devices = [dev
            for dev in devices
            for keys in (set(dev.capabilities().get(ecodes.EV_KEY, [])),)
                        if must_have.issubset(keys)
                        and must_not_have.isdisjoint(keys)]
# Pick the first keyboard
keyboard = devices[0]

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

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

14 https://docs.python.org/3.5/library/curses.html#module-curses
if event.value == 0:  # key released
    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.

2.25 Motion sensor robot

Make a robot drive forward when it detects motion:
from gpiozero import Robot, MotionSensor
from signal import pause

robot = Robot(0, 0, 1, 2)
pir = MotionSensor(5)
pir.when_motion = robot.forward
pir.when_no_motion = robot.stop
pause()

Alternatively:

from gpiozero import Robot, MotionSensor
from gpiozero.tools import zip_values
from signal import pause

robot = Robot(0, 0, 1, 2)
pir = MotionSensor(5)
robot.source = zip_values(pir, pir)

pause()

2.26 Potentiometer

Continually print the value of a potentiometer (values between 0 and 1) connected to a MCP3008 (page 135) analog to digital converter:
Present the value of a potentiometer on an LED bar graph using PWM to represent states that won’t “fill” an LED:

```python
from gpiozero import MCP3008
pot = MCP3008(channel=0)
while True:
    print(pot.value)
```

2.27 Measure temperature with an ADC

Wire a TMP36 temperature sensor to the first channel of an MCP3008 (page 135) analog to digital converter:

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

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

adc = MCP3008(channel=0)
for temp in convert_temp(adc.values):
    print('The temperature is', temp, 'C')
sleep(1)
```
2.28 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:

```python
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 176) property rather than a `while` loop:

```python
from gpiozero import RGBLED, MCP3008
from gpiozero.tools import zip_values
from signal import pause

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

(continues on next page)

---

15 https://docs.python.org/3.5/reference/compound_stmts.html#while
led.source = zip_values(red_pot, green_pot, blue_pot)
pause()

2.29 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\(^{16}\) to remotely control the lamp, and the `TimeOfDay` (page 167) class to control the timing:

```python
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
lamp.source_delay = 60
pause()
```

2.30 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 168) 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:

```python
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
green.source_delay = 60
red.source = negated(green)
pause()
```

2.31 CPU Temperature Bar Graph

You can read the Raspberry Pi’s own CPU temperature using the built-in `CPUTemperature` (page 169) class, and display this on a “bar graph” of LEDs:

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

\(^{16}\) https://energenie4u.co.uk/catalogue/product/ENER002-2PI
```python
cpu = CPUTemperature(min_temp=50, max_temp=90)
leds = LEDBarGraph(2, 3, 4, 5, 6, 7, 8, pwm=True)
leds.source = cpu
pause()
```

## 2.32 More recipes

Continue to:

- *Advanced Recipes* (page 35)
- *Remote GPIO Recipes* (page 51)
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 141)` object one-by-one:

```python
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 141)` also supports indexing. This means you can access the individual `LED (page 111)` objects using `leds[i]` where `i` is an integer from 0 up to (not including) the number of LEDs:

```python
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:

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

(continues on next 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 objects can have their LED objects named upon construction. This means the individual LEDs can be accessed by their name:

```python
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()
```

LEDBoard objects can also be nested within other LEDBoard objects:

```python
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.

```python
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
    leds.green.source_delay = 60
    leds.red.source = negated(leds.green)

pause()
```

Alternatively, using the STATUS Zero\(^\text{17}\) board:

```python
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
    leds.green.source_delay = 60
    leds.red.source = negated(leds.green)

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:

```python
from travispy import TravisPy
from gpiozero import LED
```

\(^\text{17}\) https://thepihut.com/status
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)
pause()

Note this recipe requires travispy\[^{18}\]. 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.

```python
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):
```

[^18]: https://travispy.readthedocs.io/
3.5 Robot controlled by 2 potentiometers

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

```python
from gpiozero import Robot, MCP3008
from gpiozero.tools import zip_values
from signal import pause

robot = Robot(left=(4, 14), right=(17, 18))
left_pot = MCP3008(0)
right_pot = MCP3008(1)
robot.source = zip_values(left_pot, right_pot)
pause()
```

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

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

robot = Robot(left=(4, 14), right=(17, 18))
left_pot = MCP3008(0)
right_pot = MCP3008(1)
robot.source = zip(scaled(left_pot, -1, 1), scaled(right_pot, -1, 1))
pause()
```

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

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:

```python
from bluedot import BlueDot
from gpiozero import LED

bd = BlueDot()
led = LED(17)

while True:
    bd.wait_for_press()
```

---

19 https://docs.python.org/3.5/library/functions.html#zip
Note this recipe requires `bluedot` and the associated Android app. See the BlueDot documentation\(^{20}\) 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:

```python
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:

```python
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)
```

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:

```python
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 3 the LEDs are controlled by a GPIO expander which is not accessible from gpiozero (yet).
CHAPTER 4

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\textsuperscript{21} 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 195) documentation page.

One of the pin libraries supported, pigpio\textsuperscript{22}, 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 51) 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\textsuperscript{23}.

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:

\textsuperscript{21} https://pypi.python.org/pypi/RPi.GPIO
\textsuperscript{22} http://abyz.me.uk/rpi/pigpio/python.html
\textsuperscript{23} 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
```
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 x8624), then you have everything you need. If you’re using another Linux distribution, Mac OS or Windows then you’ll need to install the pigpio25 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
```

(Alternatively, install pip with get-pip26.)

Next, install GPIO Zero and pigpio for Python 3:

25 http://abyz.me.uk/rpi/pigpio/python.html
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`²⁷. Next, install GPIO Zero and pigpio with pip:

```
$ pip3 install gpiozero pigpio
```

Or for Python 2:

```
$ pip install gpiozero pigpio
```

4.2.4 Windows

Modern Python installers for Windows bundle pip with Python. If pip is not installed, you can follow this guide²⁸. 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` (page 74) 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 pigpio²⁹ 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` (page 207). If RPi.GPIO³⁰ is installed, this will be selected as the default pin factory, so either uninstall it, or use the `GPIOZERO_PIN_FACTORY` (page 74) environment variable to override it:

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

This usage will set the pin factory to `PiGPIOFactory` (page 207) 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:

```
29 http://abyz.me.uk/rpi/pigpio/python.html
30 https://pypi.python.org/pypi/RPi.GPIO
```
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` (page 207) can be used, as pigpio\(^{31}\) 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` (page 207) objects, and use them when instantiating device objects. For example, with no environment variables set:

```python
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:

```python
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)
```

(continues on next page)

---

\(^{31}\) [http://abyz.me.uk/rpi/pigpio/python.html](http://abyz.me.uk/rpi/pigpio/python.html)
while True:
    led_1.on()
    led_2.off()
    sleep(1)
    led_1.off()
    led_2.on()
    sleep(1)

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

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

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

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

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

```python
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:

```python
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
```

(continues on next page)
while True:
    led_1.on()
    led_2.off()
    led_3.on()
    sleep(1)
    led_1.off()
    led_2.on()
    led_3.off()
sleep(1)

Note that these examples use the LED (page 111) 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, or the pin_factory keyword argument:

```python
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:

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

remote_factory = PiGPIOFactory(host='192.168.1.4')
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 HAT32) on one Pi, and use remote pins to control another over the network:

```python
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:

32 [https://www.raspberry.org/products/sense-hat/](https://www.raspberry.org/products/sense-hat/)
- Remote GPIO Recipes (page 51)
- Pi Zero USB OTG (page 55)
CHAPTER 5

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 43).

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 (page 93) on one Raspberry Pi control the LED (page 111) of another:

```python
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
pause()
```

5.2 LED + 2 Buttons

The LED (page 111) will come on when both buttons are pressed:

```python
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)
led = LED(17)
button_1 = Button(17, pin_factory=factory3)
button_2 = Button(17, pin_factory=factory4)
led.source = all_values(button_1, button_2)
pause()

5.3 Multi-room motion alert

Install a Raspberry Pi with a MotionSensor (page 97) in each room of your house, and have an class:LED indicator showing when there’s motion in each room:

```python
from gpiozero import LEDBoard, MotionSensor
from gpiozero.pins.pigpio import PiGPIOFactory
from gpiozero.tools import zip_values
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

leds.source = zip_values(*sensors)
pause()
```

5.4 Multi-room doorbell

Install a Raspberry Pi with a Buzzer (page 117) attached in each room you want to hear the doorbell, and use a push Button (page 93) as the doorbell:

```python
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

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 (page 155), 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:

```python
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\(^{33}\) (not supported by GPIO Zero) includes temperature, humidity and pressure sensors, but no light sensor. Remote GPIO allows an external `LightSensor` (page 99) to be used as well. The Sense HAT LED display can be used to show different colours according to the light levels:

```python
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.

The Raspberry Pi Zero[^34] and Pi Zero W[^35] 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 43) 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[^36] 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
```

[^34]: https://www.raspberrypi.org/products/raspberry-pi-zero/
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:

![Select Role Dialogue]

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 using the address `fe80::1%followed by the connection string. You can look this up using `ifconfig`.

Set the `GPIOZERO_PIN_FACTORY` (page 74) and `PIGPIO_ADDR` (page 74) environment variables on your PC so GPIO Zero connects to the “remote” Pi Zero:

```
$ export GPIOZERO_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:

Alternatively, you can set the pin factory in-line, as explained in Configuring Remote GPIO (page 43).
Read more on the GPIO expander in blog posts on raspberrypi.org and bennuttall.com.

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.info and learn.adafruit.com 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.

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:

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

---

37 https://www.raspberrypi.org/blog/gpio-expander/
38 http://bennuttall.com/raspberry-pi-zero-gpio-expander/
39 http://blog.gbaman.info/?p=791
40 https://learn.adafruit.com/turning-your-raspberry-pi-zero-into-a-usb-gadget/ethernet-gadget
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
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)
```

except that the former is updated in a background thread, which enables you to do other things at the same time.

Every device has a `value` (page 175) property (the device’s current value). Input devices (like buttons) can only have their values read, but output devices (like LEDs) 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
```

(continues on next page)
Every device also has a `values` (page 176) property (a generator\(^{41}\) continuously yielding the device’s current value). All output devices have a `source` (page 176) property which can be set to any iterator\(^{42}\). The device will iterate over the values of the device provided, setting the device’s value to each element at a rate specified in the `source_delay` (page 176) property (the default is 0.01 seconds).

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

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

led = PWMLED(17)
pot = MCP3008()

led.source = pot

pause()
```

The way this works is that the input device’s `values` (page 176) property is used to feed values into the output device. Prior to v1.5, the `source` (page 176) had to be set directly to a device’s `values` (page 176) property:

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

led = PWMLED(17)
pot = MCP3008()

led.source = pot.values

pause()
```

**Note:** Although this method is still supported, the recommended way is now to set the `source` (page 176) to a device object.

It is also possible to set an output device’s `source` (page 176) to another output device, to keep them matching. In this example, the red LED is set to match the button, and the green LED is set to match the red LED, so both LEDs will be on when the button is pressed:

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

led = PWMLED(17)
pot = MCP3008()

led.source = pot.values

pause()
```

\(^{41}\) [https://wiki.python.org/moin/Generators]

\(^{42}\) [https://wiki.python.org/moin/Iterator]
from gpiozero import LED, Button
from signal import pause
red = LED(14)
green = LED(15)
button = Button(17)
red.source = button
green.source = red
pause()

7.1 Processing values

The device’s values can also be processed before they are passed to the source (page 176):

For example, writing a generator function to pass the opposite of the Button value into the LED:

from gpiozero import Button, LED
from signal import pause
def opposite(device):
    for value in device.values:
        yield not value
led = LED(4)
btn = Button(17)
led.source = opposite(btn)
pause()

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

For example, writing a generator function to randomly yield 0 or 1:

from gpiozero import LED
from random import randint
from signal import pause
def rand():
(continues on next page)
If the iterator is infinite (i.e. an infinite generator), the elements will be processed until the `source` (page 176) 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):

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

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

### 7.2 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 179).

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:

```python
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()
```

Note that in the above example, `source_delay` (page 176) is used to make the LED iterate over the random values slightly slower. `source_delay` (page 176) can be set to a larger number (e.g. 1 for a one second delay) or set to 0 to disable any delay.

Some tools take a single source and process its values:

---

43 [https://docs.python.org/3.5/library/constants.html#None](https://docs.python.org/3.5/library/constants.html#None)
In this example, the LED is lit only when the button is not pressed:

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

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

led.source = negated(btn)
pause()
```

**Note:** Note that source tools which take one or more value parameters support passing either `ValuesMixin` (page 176) derivatives, or iterators, including a device’s `values` (page 176) property.

Some tools combine the values of multiple sources:

```python
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, button_b)
pause()
```

In this example, the LED is lit only if both buttons are pressed (like an AND\(^{44}\) gate):


---

**7.2. Source Tools**
Similarly, `any_values()` (page 183) with two buttons would simulate an OR gate.

While most devices have a value (page 175) range between 0 and 1, some have a range between -1 and 1 (e.g. `Motor` (page 120), `Servo` (page 123) and `TonalBuzzer` (page 119)). Some source tools output values between -1 and 1, which are ideal for these devices, for example passing `sin_values()` (page 185) in:

```
from gpiozero import Motor, Servo, TonalBuzzer
from gpiozero.tools import sin_values
from signal import pause

motor = Motor(2, 3)
servo = Servo(4)
buzzer = TonalBuzzer(5)

motor.source = sin_values()
servo.source = motor
buzzer.source = motor

pause()
```

In this example, all three devices are following the sine wave. The motor value ramps up from 0 (stopped) to 1 (full speed forwards), then back down to 0 and on to -1 (full speed backwards) in a cycle. Similarly, the servo moves from its mid point to the right, then towards the left; and the buzzer starts with its mid tone, gradually raises its frequency, to its highest tone, then down towards its lowest tone. Note that setting `source_delay` (page 176) will alter the speed at which the device iterates through the values. Alternatively, the tool `cos_values()` (page 185) could be used to start from -1 and go up to 1, and so on.

### 7.3 Internal devices

GPIO Zero also provides several internal devices (page 167) 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. These classes include a values (page 176) property which can be used to feed values into a device’s source (page 176). For example, a lamp connected to an Energy (page 159) socket can be controlled by a `TimeOfDay` (page 167) object so that it is on between the hours of 8am and 8pm:

```
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
```

---

45 https://en.wikipedia.org/wiki/OR_gate

lamp.source_delay = 60
pause()

Using the `DiskUsage` (page 171) class with `LEDBarGraph` (page 144) can show your Pi’s disk usage percentage on a bar graph:

```python
from gpiozero import DiskUsage, LEDBarGraph
from signal import pause
disk = DiskUsage()
graph = LEDBarGraph(2, 3, 4, 5, 6, 7, 8)
graph.source = disk
pause()
```

Demonstrating a garden light system whereby the light comes on if it’s dark and there’s motion is simple enough, but it requires using the `booleanized()` (page 179) source tool to convert the light sensor from a float value into a boolean:

```python
from gpiozero import LED, MotionSensor, LightSensor
from gpiozero.tools import booleanized, all_values
from signal import pause
garden = LED(2)
motion = MotionSensor(4)
light = LightSensor(5)
garden.source = all_values(booleanized(light, 0, 0.1), motion)
pause()
```

7.4 Composite devices

The `value` (page 175) of a composite device made up of the nested values of its devices. For example, the value of a `Robot` (page 155) object is a 2-tuple containing its left and right motor values:

```python
>>> 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)
```
Use two potentiometers to control the left and right motor speed of a robot:

\[
\text{Robot} \quad \text{zip\_values} \quad \text{Left potentiometer} \\
\text{Right potentiometer}
\]

```python
from gpiozero import Robot, MCP3008
from gpiozero.tools import zip_values
from signal import pause

robot = Robot(left=(4, 14), right=(17, 18))
left_pot = MCP3008(0)
right_pot = MCP3008(1)

robot.source = zip_values(left_pot, right_pot)
pause()
```

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

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

robot = Robot(left=(4, 14), right=(17, 18))
left_pot = MCP3008(0)
right_pot = MCP3008(1)

robot.source = zip(scaled(left_pot, -1, 1), scaled(right_pot, -1, 1))
pause()
```

Note that this example uses the built-in `zip()` rather than the tool `zip_values()` (page 184) as the `scaled()` (page 182) tool yields values which do not need converting, just zipping. Also note that this use

---

47 https://docs.python.org/3.5/library/functions.html#zip
of `zip()`\(^{48}\) will not work in Python 2, instead use `izip`\(^{49}\).

\(^{48}\) https://docs.python.org/3.5/library/functions.html#zip
\(^{49}\) https://docs.python.org/2/library/itertools.html#itertools.izip
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

For further information, please refer to https://pinout.xyz/
8.1.1 Synopsis

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

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 43)) 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 71)

- `-m`, `--monochrome`
  Force monochrome output. See also `--color` (page 71)

- `-x`, `--xyz`
  Open pinout.xyz\(^{50}\) in the default web browser

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:

```
,----------------------------------
| ooooooooooooooooooooooooooooo J8 +====
| 1ooooooooooooooooooo | USB +====
| +====
| Pi Model 3B V1.1 |
| +-----+ +====
| |D| |SoC | | USB +====
| |S| | |
| |I| +------
| |C| +------
| pwr |HDMI| |I| |A| +------`

Revision : a02082
SoC : BCM2837
RAM : 1024Mb
Storage : MicroSD
USB ports : 4 (excluding power)
Ethernet ports : 1
Wi-fi : True
```

\(^{50}\) https://pinout.xyz/
Bluetooth : True
Camera ports (CSI) : 1
Display ports (DSI): 1

J8:
  3V3 (1) (2) 5V
  GPIO2 (3) (4) 5V
  GPIO3 (5) (6) GND
  GPIO4 (7) (8) GPIO14
    GND (9) (10) GPIO15
  GPIO17 (11) (12) GPIO18
  GPIO27 (13) (14) GND
  GPIO22 (15) (16) GPIO23
    3V3 (17) (18) GPIO24
  GPIO10 (19) (20) GND
  GPIO9 (21) (22) GPIO25
  GPIO11 (23) (24) GPIO8
    GND (25) (26) GPIO7
  GPIO6 (27) (28) GPIO1
  GPIO5 (29) (30) GND
  GPIO13 (31) (32) GPIO12
  GPIO19 (33) (34) GND
  GPIO26 (35) (36) 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 71):

```
$ pinout --monochrome
```

Or forced to be --color (page 71), 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 71). The tool understands both old-style revision codes\(^1\) (such as for the model B):

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

Or new-style revision codes\(^2\) (such as for the Pi Zero W):

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

---


You can also use the tool with `Configuring Remote GPIO` (page 43) to query remote Raspberry Pi’s:

```bash
$ 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):

8.1. pinout
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.
9.1 How do I keep my script running?

The following script looks like it should turn an LED (page 111) on:

```python
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()` ⁵³ to keep the script alive:

```python
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:

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

def hello():
    print("Hello")

button = Button(2)
button.when_pressed = hello

pause()
```

⁵³ 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:

```python
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` (page 95), the thing that is assigned is the result of calling the `pushed` function. Because `pushed` doesn’t explicitly return anything, the result is `None`\(^54\). Hence this is equivalent to doing:

```python
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:

```python
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).

**Note:** Note that as of v1.5, setting a callback to `None` when it was previously `None` will raise a `CallbackSetToNone` (page 216) warning, with the intention of alerting users when callbacks are set to `None` accidentally. However, if this is intentional, the warning can be suppressed. See the `warnings`\(^55\) module for reference.

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:

```bash
$ 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 195)). 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 the `GPIOZERO_PIN_FACTORY` (page 74) environment variable. For example:

   ```bash
   $ GPIOZERO_PIN_FACTORY=pigpio python3
   ```

\(^54\) [https://docs.python.org/3.5/library/constants.html#None](https://docs.python.org/3.5/library/constants.html#None)

\(^55\) [https://docs.python.org/3.5/library/warnings.html#module-warnings](https://docs.python.org/3.5/library/warnings.html#module-warnings)
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`\(^{56}\) will be raised.

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

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

Refer to the `warnings`\(^{57}\) 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:

```python
>>> from pkg_resources import require
>>> require('gpiozero')
[gpiozero 1.5.0 (/usr/lib/python3/dist-packages)]
>>> require('gpiozero')[0].version
'1.5.0'
```

If you have multiple versions installed (e.g. from `pip` and `apt`) they will not show up in the list returned by the `pkg_resources.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`\(^{58}\).

### 9.5 Why do I get “command not found” when running pinout?

The gpiozero library is available as a Debian package for Python 2 and Python 3, but the `cli_pinout` tool cannot be made available by both packages, so it’s only included with the Python 3 version of the package. To make sure the `cli_pinout` tool is available, the “python3-gpiozero” package must be installed:

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

Alternatively, installing gpiozero using `pip` will install the command line tool, regardless of Python version:

```
$ sudo pip3 install gpiozero
```

or:

```
$ sudo pip install gpiozero
```

---

\(^{56}\) [https://docs.python.org/3.5/library/exceptions.html#ImportError](https://docs.python.org/3.5/library/exceptions.html#ImportError)

\(^{57}\) [https://docs.python.org/3.5/library/warnings.html#module-warnings](https://docs.python.org/3.5/library/warnings.html#module-warnings)

9.6 The pinout command line tool incorrectly identifies my Raspberry Pi model

If your Raspberry Pi model is new, it’s possible it wasn’t known about at the time of the gpiozero release you are using. Ensure you have the latest version installed (remember, the cli_pinout tool usually comes from the Python 3 version of the package as noted in the previous FAQ).

If the Pi model you are using isn’t known to gpiozero, it may have been added since the last release. You can check the GitHub issues[59] to see if it’s been reported before, or check the commits[60] on GitHub since the last release to see if it’s been added. The model determination can be found in gpiozero/pins/data.py.

9.7 What’s the gpiozero equivalent of GPIO.cleanup()?

Many people ask how to do the equivalent of the cleanup function from RPi.GPIO. In gpiozero, at the end of your script, cleanup is run automatically, restoring your GPIO pins to the state they were found.

To explicitly close a connection to a pin, you can manually call the close() (page 175) method on a device object:

```python
>>> led = LED(2)
>>> led.on()
>>> led
<gpiozero.LED object on pin GPIO2, active_high=True, is_active=True>
>>> led.close()
>>> led
<gpiozero.LED object closed>
```

This means that you can reuse the pin for another device, and that despite turning the LED on (and hence, the pin high), after calling close() (page 175) it is restored to its previous state (LED off, pin low).

9.8 How do I use button.when_pressed and button.when_held together?

The Button (page 93) class provides a when_held (page 95) property which is used to set a callback for when the button is held down for a set amount of time (as determined by the hold_time (page 95) property). If you want to set when_held (page 95) as well as when_pressed (page 95), you’ll notice that both callbacks will fire. Sometimes, this is acceptable, but often you’ll want to only fire the when_pressed (page 95) callback when the button has not been held, only pressed.

The way to achieve this is to not set a callback on when_pressed (page 95), and instead use when_released (page 95) to work out whether it had been held or just pressed:

```python
from gpiozero import Button

Button.was_held = False

def held(btn):
    btn.was_held = True
    print("button was held not just pressed")

def released(btn):
    if not btn.was_held:
        pressed()
```

[59] https://github.com/RPi-Distro/python-gpiozero/issues
[60] https://github.com/RPi-Distro/python-gpiozero/commits/master
9.9 Why do I get “ImportError: cannot import name” when trying to import from gpiozero?

It’s common to see people name their first gpiozero script `gpiozero.py`. Unfortunately, this will cause your script to try to import itself, rather than the gpiozero library from the libraries path. You’ll see an error like this:

```
Traceback (most recent call last):
  File "gpiozero.py", line 1, in <module>
    from gpiozero import LED
  File "/home/pi/gpiozero.py", line 1, in <module>
    from gpiozero import LED
ImportError: cannot import name 'LED'
```

Simply rename your script to something else, and run it again. Be sure not to name any of your scripts the same name as a Python module you may be importing, such as `picamera.py`.

9.10 Why do I get an AttributeError trying to set attributes on a device object?

If you try to add an attribute to a gpiozero device object after its initialization, you’ll find you can’t:

```
>>> from gpiozero import Button
>>> btn = Button(2)
>>> btn.label = 'alarm'
Traceback (most recent call last):
  File "stdin", line 1, in <module>
  File "/usr/lib/python3/dist-packages/gpiozero/devices.py", line 118, in __setattr__
    self.__class__.__name__, name))
AttributeError: 'Button' object has no attribute 'label'
```

This is in order to prevent users accidentally setting new attributes by mistake. Because gpiozero provides functionality through setting attributes via properties, such as callbacks on buttons (and often there is no immediate feedback when setting a property), this could lead to bugs very difficult to find. Consider the following example:

```
from gpiozero import Button

def hello():
    print("hello")

btn = Button(2)
btn.pressed = hello
```
This is perfectly valid Python code, and no errors would occur, but the program would not behave as expected: pressing the button would do nothing, because the property for setting a callback is `when_pressed` not `pressed`. But without `gpiozero` preventing this non-existent attribute from being set, the user would likely struggle to see the mistake.

If you really want to set a new attribute on a device object, you need to create it in the class before initializing your object:

```python
>>> from gpiozero import Button
>>> Button.label = ''
>>> btn = Button(2)
>>> btn.label = 'alarm'
>>> def press(btn):
...     print(btn.label, "was pressed")
>>> btn.when_pressed = press
```

### 9.11 Why is it called GPIO Zero? Does it only work on Pi Zero?

gpiozero works on all Raspberry Pi models, not just the Pi Zero.

The “zero” is part of a naming convention for “zero-boilerplate” education friendly libraries, which started with Pygame Zero\(^\text{61}\), and has been followed by NetworkZero\(^\text{62}\), guizero\(^\text{63}\) and more.

These libraries aim to remove barrier to entry and provide a smooth learning curve for beginners by making it easy to get started and easy to build up to more advanced projects.

---


\(^\text{63}\) [https://lawsie.github.io/guizero/](https://lawsie.github.io/guizero/)
CHAPTER 10

Migrating from RPi.GPIO

If you are familiar with the RPi.GPIO library, you will be used to writing code which deals with pins and the state of pins. You will see from the examples in this documentation that we generally refer to things like LEDs and Buttons rather than input pins and output pins.

GPIO Zero provides classes which represent devices, so instead of having a pin number and telling it to go high, you have an LED and you tell it to turn on, and instead of having a pin number and asking if it’s high or low, you have a button and ask if it’s pressed. There is also no boilerplate code to get started — you just import the parts you need.

GPIO Zero provides many device classes, each with specific methods and properties bespoke to that device. For example, the functionality for an HC-SR04 Distance Sensor can be found in the DistanceSensor class.

As well as specific device classes, we provide base classes InputDevice (page 107) and OutputDevice (page 130). One main difference between these and the equivalents in RPi.GPIO is that they are classes, not functions, which means that you initialize one to begin, and provide its pin number, but then you never need to use the pin number again, as it’s stored by the object.

GPIO Zero was originally just a layer on top of RPi.GPIO, but we later added support for various other underlying pin libraries. RPi.GPIO is currently the default pin library used. Read more about this in Changing the pin factory (page 196).

10.1 Output devices

Turning an LED on in RPi.GPIO:

```python
import RPi.GPIO as GPIO
GPIO.setmode(GPIO.BCM)
GPIO.setwarnings(False)
GPIO.setup(2, GPIO.OUT)
GPIO.output(2, GPIO.HIGH)
```

---

64 https://pypi.org/project/RPi.GPIO/
65 https://pypi.org/project/RPi.GPIO/
Turning an LED on in GPIO Zero:

```python
from gpiozero import LED
led = LED(2)
led.on()
```

The `LED` class also supports threaded blinking through the `blink()` method.

`OutputDevice` is the base class for output devices, and can be used in a similar way to output devices in RPi.GPIO.

See a full list of supported output devices. Other output devices have similar property and method names. There is commonality in naming at base level, such as `OutputDevice.is_active`, which is aliased in a device class, such as `LED.is_lit`.

### 10.2 Input devices

Reading a button press in RPi.GPIO:

```python
import RPi.GPIO as GPIO
GPIO.setmode(GPIO.BCM)
GPIO.setwarnings(False)
GPIO.setup(4, GPIO.IN, GPIO.PUD_UP)
if not GPIO.input(4):
    print("button is pressed")
```

Reading a button press in GPIO Zero:

```python
from gpiozero import Button
btn = Button(4)
if btn.is_pressed:
    print("button is pressed")
```

Note that in the RPi.GPIO example, the button is set up with the option `GPIO.PUD_UP` which means “pull-up”, and therefore when the button is not pressed, the pin is high. When the button is pressed, the pin goes low, so the condition requires negation (`if not`). If the button was configured as pull-down, the logic is reversed and the condition would become `if GPIO.input(4):

```python
import RPi.GPIO as GPIO
GPIO.setmode(GPIO.BCM)
GPIO.setwarnings(False)
GPIO.setup(4, GPIO.IN, GPIO.PUD_DOWN)
if GPIO.input(4):
    print("button is pressed")
```

In GPIO Zero, the default configuration for a button is pull-up, but this can be configured at initialization, and the rest of the code stays the same:

---

66 https://pypi.org/project/RPi.GPIO/
from gpiozero import Button
btn = Button(4, pull_up=False)
if btn.is_pressed:
    print("button is pressed")

RPi.GPIO also supports blocking edge detection.
Wait for a pull-up button to be pressed in RPi.GPIO:

import RPi.GPIO as GPIO
GPIO.setmode(GPIO.BCM)
GPIO.setwarnings(False)
GPIO.setup(4, GPIO.IN, GPIO.PUD_UP)
GPIO.wait_for_edge(4, GPIO.FALLING):
    print("button was pressed")

The equivalent in GPIO Zero:

from gpiozero import Button
btn = Button(4)
btn.wait_for_press()
print("button was pressed")

Again, if the button is pulled down, the logic is reversed. Instead of waiting for a falling edge, we’re waiting for a rising edge:

import RPi.GPIO as GPIO
GPIO.setmode(GPIO.BCM)
GPIO.setwarnings(False)
GPIO.setup(4, GPIO.IN, GPIO.PUD_UP)
GPIO.wait_for_edge(4, GPIO.FALLING):
    print("button was pressed")

Again, in GPIO Zero, the only difference is in the initialization:

from gpiozero import Button
btn = Button(4, pull_up=False)
btn.wait_for_press()
print("button was pressed")

RPi.GPIO has threaded callbacks. You create a function (which must take one argument), and pass it in to add_event_detect, along with the pin number and the edge direction:

import RPi.GPIO as GPIO
GPIO.setmode(GPIO.BCM)
GPIO.setwarnings(False)
def pressed(pin):
    print("button was pressed")
In GPIO Zero, you assign the `when_pressed` (page 95) and `when_released` (page 95) properties to set up callbacks on those actions:

```python
from gpiozero import Buttons

def pressed():
    print("button was pressed")

def released():
    print("button was released")

btn = Button(4)
btn.when_pressed = pressed
btn.when_released = released
```

`when_held` (page 95) is also provided, where the length of time considered a “hold” is configurable.

The callback functions don’t have to take any arguments, but if they take one, the button object is passed in, allowing you to determine which button called the function.

`InputDevice` (page 107) is the base class for input devices, and can be used in a similar way to input devices in RPi.GPIO.

See a full list of input devices (page 93). Other input devices have similar property and method names. There is commonality in naming at base level, such as `InputDevice.is_active` (page 108), which is aliased in a device class, such as `Button.is_pressed` (page 95) and `LightSensor.light_detected` (page 100).

## 10.3 Composite devices, boards and accessories

Some devices require connections to multiple pins, for example a distance sensor, a combination of LEDs or a HAT. Some GPIO Zero devices comprise multiple device connections within one object, such as `RGBLED` (page 115), `LEDBoard` (page 141), `DistanceSensor` (page 101), `Motor` (page 120) and `Robot` (page 155).

With RPi.GPIO, you would have one output pin for the trigger, and one input pin for the echo. You would time the echo and calculate the distance. With GPIO Zero, you create a single `DistanceSensor` (page 101) object, specifying the trigger and echo pins, and you would read the `DistanceSensor.distance` (page 102) property which automatically calculates the distance within the implementation of the class.

The `Motor` (page 120) class controls two output pins to drive the motor forwards or backwards. The `Robot` (page 155) class controls four output pins (two motors) in the right combination to drive a robot forwards or backwards, and turn left and right.

The `LEDBoard` (page 141) class takes an arbitrary number of pins, each controlling a single LED. The resulting `LEDBoard` (page 141) object can be used to control all LEDs together (all on / all off), or individually by index. Also the object can be iterated over to turn LEDs on in order. See examples of this (including slicing) in the `advanced recipes` (page 35).
10.4 PWM (Pulse-width modulation)

Both libraries support software PWM control on any pin. Depending on the pin library used, GPIO Zero can also support hardware PWM (using RPIOPin or PiGPIOPin).

A simple example of using PWM is to control the brightness of an LED.

In RPi.GPIO:

```python
import RPi.GPIO as GPIO
from time import sleep
GPIO.setmode(GPIO.BCM)
GPIO.setwarnings(False)
GPIO.setup(2, GPIO.OUT)
pwm = GPIO.PWM(2, 100)
pwm.start(0)
for dc in range(100):
    pwm.changeDutyCycle(dc)
sleep(0.01)
```

In GPIO Zero:

```python
from gpiozero import PWMLED
from time import sleep
led = PWMLED(2)
for b in range(100):
    led.value = b / 100
    sleep(0.01)
```

PWMLED (page 113) has a `blink()` (page 113) method which can be used the same was as LED (page 111)’s `blink()` (page 112) method, but its PWM capabilities allow for `fade_in` and `fade_out` options to be provided. There is also the `pulse()` (page 114) method which provides a neat way to have an LED fade in and out repeatedly.

Other devices can make use of PWM, such as motors (for variable speed) and servos. See the Motor (page 120), Servo (page 123) and AngularServo (page 124) classes for information on those. Motor (page 120) and Robot (page 155) default to using PWM, but it can be disabled with `pwm=False` at initialization. Servos cannot be used without PWM. Devices containing LEDs default to not using PWM, but `pwm=True` can be specified and any LED objects within the device will be initialized as PWMLED (page 113) objects.

10.5 Cleanup

Pin state cleanup is explicit in RPi.GPIO, and is done manually with `GPIO.cleanup()` but in GPIO Zero, cleanup is automatically performed on every pin used, at the end of the script. Manual cleanup is possible by use of the `close()` (page 175) method on the device.

Read more in the relevant FAQ: What’s the gpiozero equivalent of GPIO.cleanup()? (page 78)

10.6 Pi Information

RPi.GPIO provides information about the Pi you’re using. The equivalent in GPIO Zero is the function `pi_info()` (page 189):

---

67 https://pypi.org/project/RPi.GPIO/
>>> from gpiozero import pi_info
>>> pi = pi_info()
>>> pi
PiBoardInfo(revision='a02082', model='3B', pcb_revision='1.2', released='2016Q1',
→soc='BCM2837', manufacturer='Sony', memory=1024, storage='MicroSD', usb=4,
→ethernet=1, wifi=True, bluetooth=True, csi=1, dsi=1, headers=..., board=...)
>>> pi.soc
'BCM2837'
>>> pi.wifi
True

Read more about what PiBoardInfo (page 189) provides.

## 10.7 More

GPIO Zero provides more than just GPIO device support, it includes some support for SPI devices (page 133) including a range of analog to digital converters.

Device classes which are compatible with other GPIO devices, but have no relation to GPIO pins, such as CPUTemperature (page 169), TimeOfDay (page 167), PingServer (page 168) and LoadAverage (page 170) are also provided.

GPIO Zero features support for multiple pin libraries. The default is to use RPi.GPIO to control the pins, but you can choose to use another library, such as pigpio, which supports network controlled GPIO. See Changing the pin factory (page 196) and Configuring Remote GPIO (page 43) for more information.

It is possible to run GPIO Zero on your PC, both for remote GPIO and for testing purposes, using Mock pins (page 197).

Another feature of this library is configuring devices to be connected together in a logical way, for example in one line you can say that an LED and button are “paired”, i.e. the button being pressed turns the LED on. Read about this in Source/Values (page 59).

## 10.8 FAQs

Note the following FAQs which may catch out users too familiar with RPi.GPIO:

- *How do I keep my script running?* (page 75)
- *Why do I get PinFactoryFallback warnings when I import gpiozero?* (page 76)
- *What’s the gpiozero equivalent of GPIO.cleanup()?* (page 78)
Contributing

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

11.1 Suggestions

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

11.2 Bugs

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

11.3 Documentation

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

11.4 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.

Copyrights on submissions are owned by their authors (we don’t bother with copyright assignments), and we assume that authors are happy for their code to be released under the project’s license (page 225). Do feel free

---

68 https://github.com/RPi-Distro/python-gpiozero/issues
69 https://github.com/RPi-Distro/python-gpiozero/issues
70 https://github.com/RPi-Distro/python-gpiozero/tree/master/docs
to add your name to the list of contributors in README.rst at the top level of the project in your pull request! Don’t worry about adding your name to the copyright headers in whatever files you touch; these are updated automatically from the git metadata before each release.

### 11.5 Backwards compatibility

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

### 11.6 Python 2/3

The library is 100% compatible with both Python 2.7 and Python 3 from version 3.2 onwards. We intend to drop Python 2 support in 2020 when Python 2 reaches end-of-life\(^7\).

\(^7\) 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 111) and follow its heritage backward to DigitalOutputDevice (page 127). Follow that back to OutputDevice (page 130) 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).

12.1 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:

```bash
$ sudo apt install lsb-release build-essential git git-core \
  exuberant-ctags virtualenvwrapper python-virtualenv python3-virtualenv \
  python-dev python3-dev
```

After installing virtualenvwrapper you’ll need to restart your shell before commands like `mkvirtualenv` will operate correctly. Once you’ve restarted your shell, continue:

```bash
$ 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 usable at this stage, but doesn’t support facilities like PWM):
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 which limits bandwidth):

```
(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
```

### 12.2 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 python-sphinx latexmk
```

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 `build/html` while the PDF output goes to `build/latex`.

### 12.3 Test suite

If you wish to run the GPIO Zero test suite, follow the instructions in Development installation (page 89) above and then make the “test” target within the sandbox. You’ll also need to install some pip packages:

```
$ workon python-gpiozero
(python-gpiozero) $ pip install coverage mock pytest
(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 1KΩ 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!).
The test suite is also setup for usage with the `tox` utility, in which case it will attempt to execute the test suite with all supported versions of Python. If you are developing under Ubuntu you may wish to look into the Dead Snakes PPA\(^{72}\) in order to install old/new versions of Python; the tox setup *should* work with the version of tox shipped with Ubuntu Xenial, but more features (like parallel test execution) are available with later versions.

On the subject of parallel test execution, this is also supported in the tox setup, including the “real” pin tests (a file-system level lock is used to ensure different interpreters don’t try to access the physical pins simultaneously).

For example, to execute the test suite under tox, skipping interpreter versions which are not installed:

```
$ tox -s
```

To execute the test suite under all installed interpreter versions in parallel, using as many parallel tasks as there are CPUs, then displaying a combined report of coverage from all environments:

```
$ tox -p auto -s
$ coverage combine --rcfile coverage.cfg
$ coverage report --rcfile coverage.cfg
```

\(^{72}\) https://launchpad.net/~deadsnakes/+Archive/ubuntu/ppa
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 by default. See the *Pin Numbering* (page 3) section for more information.

### 13.1 Regular Classes

The following classes are intended for general use with the devices they represent. All classes in this section are concrete (not abstract).

#### 13.1.1 Button

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

Extends `DigitalInputDevice` (page 104) and represents a simple push button or switch.

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

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

```python
from gpiozero import Button

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

**Parameters**

73 https://docs.python.org/3.5/library/constants.html#False
- **pin** (*int* or *str*) – The GPIO pin which the button is connected to. See *Pin Numbering* (page 3) for valid pin numbers. If this is *None* a *GPIODeviceError* (page 213) will be raised.

- **pull_up** (*bool* or *None*) – 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. If *None*, the pin will be floating, so it must be externally pulled up or down and the *active_state* parameter must be set accordingly.

- **active_state** (*bool* or *None*) – See description under *InputDevice* (page 107) for more information.

- **bounce_time** (*float* or *None*) – If *None* (the default), no software bounce compensation will be performed. Otherwise, this is the length of time (in seconds) that the component will ignore changes in state after an initial change.

- **hold_time** (*float*) – The length of time (in seconds) to wait after the button is pushed, until executing the *when_held* (page 95) handler. Defaults to 1.

- **hold_repeat** (*bool*) – If *True*, the *when_held* (page 95) handler will be repeatedly executed as long as the device remains active, every *hold_time* seconds. If *False* (the default) the *when_held* (page 95) handler will be only be executed once per hold.

- **pin_factory** (*Factory* or *None*) – See *API - Pins* (page 195) for more information (this is an advanced feature which most users can ignore).

```python
wait_for_press(timeout=None)
```

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

**Parameters**

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

```python
wait_for_release(timeout=None)
```

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

**Parameters**

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

```python
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 95) event rather than when the device activated, in contrast to `active_time` (page 177). If the device is not currently held, this is `None`\(^{98}\).

**hold_repeat**

If `True`\(^{99}\), `when_held` (page 95) will be executed repeatedly with `hold_time` (page 95) 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 95) handler. If `hold_repeat` (page 95) is True, this is also the length of time between invocations of `when_held` (page 95).

**is_held**

When `True`\(^{100}\), the device has been active for at least `hold_time` (page 95) seconds.

**is_pressed**

Returns `True`\(^{101}\) if the device is currently active and `False`\(^{102}\) otherwise. This property is usually derived from `value` (page 95). Unlike `value` (page 95), this is *always* a boolean.

**pin**

The `Pin` (page 199) that the device is connected to. This will be `None`\(^{103}\) if the device has been closed (see the `close()` (page 175) 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`\(^{104}\), the device uses a pull-up resistor to set the GPIO pin “high” by default.

**value**

Returns 1 if the button is currently pressed, and 0 if it is not.

**when_held**

The function to run when the device has remained active for `hold_time` (page 95) 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`\(^{105}\) (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`\(^{106}\) (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`\(^{107}\) (the default) to disable the event.

---

\(^{98}\) https://docs.python.org/3.5/library/constants.html#None

\(^{99}\) https://docs.python.org/3.5/library/constants.html#True

\(^{100}\) https://docs.python.org/3.5/library/constants.html#True

\(^{101}\) https://docs.python.org/3.5/library/constants.html#False

\(^{102}\) https://docs.python.org/3.5/library/constants.html#None

\(^{103}\) https://docs.python.org/3.5/library/constants.html#None

\(^{104}\) https://docs.python.org/3.5/library/constants.html#True

\(^{105}\) https://docs.python.org/3.5/library/constants.html#None

\(^{106}\) https://docs.python.org/3.5/library/constants.html#None

\(^{107}\) https://docs.python.org/3.5/library/constants.html#None
13.1.2 LineSensor (TRCT5000)

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

Extends SmoothedInputDevice (page 105) and represents a single pin line sensor like the TCRT5000 infra-red proximity sensor found in the CamJam #3 EduKit.

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

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

```python
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` or `str` – The GPIO pin which the sensor is connected to. See Pin Numbering (page 3) for valid pin numbers. If this is `None` a `GPIODeviceError` (page 213) will be raised.

- **pull_up**: `bool` or `None` – See description under InputDevice (page 107) for more information.

- **active_state**: `bool` or `None` – See description under InputDevice (page 107) for more information.

- **queue_len**: `int` – The length of the queue used to store values read from the sensor. This defaults to 5.

- **sample_rate**: `float` – The number of values to read from the device (and append to the internal queue) per second. Defaults to 100.

- **threshold**: `float` – Defaults to 0.5. When the average of all values in the internal queue rises above this value, the sensor will be considered “active” by the `is_active` (page 106) property, and all appropriate events will be fired.

- **partial**: `bool` – When `False` (the default), the object will not return a value for `is_active` (page 106) until the internal queue has filled with values. Only set this to `True` if you require values immediately after object construction.

- **pin_factory**: `Factory` or `None` – See API - Pins (page 195) for more information (this is an advanced feature which most users can ignore).
wait_for_line (timeout=None)
Pauses the script until the device is deactivated, or the timeout is reached.

Parameters

timeout (float or None) – 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)
Pauses the script until the device is activated, or the timeout is reached.

Parameters

timeout (float or None) – 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 199) that the device is connected to. This will be None if the device has been closed (see the close() (page 175) 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 average of the queued values. This is nearer 0 for black under the sensor, and nearer 1 for white under the sensor.

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.

13.1.3 MotionSensor (D-SUN PIR)

class gpiozero.MotionSensor (pin, *, queue_len=1, sample_rate=10, threshold=0.5, partial=False, pin_factory=None)

Extends SmoothedInputDevice (page 105) and represents a passive infra-red (PIR) motion sensor like the sort found in the CamJam #2 EduKit.

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

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

```python
from gpiozero import MotionSensor
pir = MotionSensor(4)
```

(continues on next page)
pir.wait_for_motion()
print("Motion detected!")

Parameters

• **pin** *(int\(^{133}\) or str\(^{134}\)) – The GPIO pin which the sensor is connected to. See Pin Numbering (page 3) for valid pin numbers. If this is None\(^{135}\) a GPIODeviceError (page 213) will be raised.*

• **pull_up** *(bool\(^{136}\) or None\(^{137}\)) – See description under InputDevice (page 107) for more information.*

• **active_state** *(bool\(^{138}\) or None\(^{139}\)) – See description under InputDevice (page 107) for more information.*

• **queue_len** *(int\(^{140}\)) – 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\(^{141}\)) – The number of values to read from the device (and append to the internal queue) per second. Defaults to 100.*

• **threshold** *(float\(^{142}\)) – Defaults to 0.5. When the average of all values in the internal queue rises above this value, the sensor will be considered “active” by the is_active (page 106) property, and all appropriate events will be fired.*

• **partial** *(bool\(^{143}\)) – When False\(^{144}\) (the default), the object will not return a value for is_active (page 106) until the internal queue has filled with values. Only set this to True\(^{145}\) if you require values immediately after object construction.*

• **pin_factory** *(Factory (page 198) or None\(^{146}\)) – See API - Pins (page 195) 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\(^{147}\) or None\(^{148}\)) – Number of seconds to wait before proceeding. If this is None\(^{149}\) (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\(^{150}\) or None\(^{151}\)) – Number of seconds to wait before proceeding. If this is None\(^{152}\) (the default), then wait indefinitely until the device is inactive.*
motion_detected
Returns `True`\(^{133}\) if the `value` (page 107) currently exceeds `threshold` (page 107) and `False`\(^{154}\) otherwise.

pin
The `Pin` (page 199) that the device is connected to. This will be `None`\(^{155}\) if the device has been closed (see the `close()` (page 175) method). When dealing with GPIO pins, query `pin.number` to discover the GPIO pin (in BCM numbering) that the device is connected to.

value
With the default `queue_len` of 1, this is effectively boolean where 0 means no motion detected and 1 means motion detected. If you specify a `queue_len` greater than 1, this will be an averaged value where values closer to 1 imply motion detection.

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

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

Set this property to `None`\(^{156}\) (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`\(^{157}\) (the default) to disable the event.

13.1.4 LightSensor (LDR)

```python
class gpiozero.LightSensor(pin, *, queue_len=5, charge_time_limit=0.01, threshold=0.1, partial=False, pin_factory=None)
```

Extends `SmoothedInputDevice` (page 105) and represents a light dependent resistor (LDR).

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

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

```python
from gpiozero import LightSensor

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

Parameters

- `pin`\(^{158}\) or `str`\(^{159}\) – The GPIO pin which the sensor is attached to. See `Pin Numbering` (page 3) for valid pin numbers. If this is `None`\(^{160}\) a `GPIODeviceError` (page 213) will be raised.

\(^{133}\) https://docs.python.org/3.5/library/constants.html#True

\(^{154}\) https://docs.python.org/3.5/library/constants.html#False

\(^{155}\) https://docs.python.org/3.5/library/constants.html#None

\(^{156}\) https://docs.python.org/3.5/library/constants.html#None

\(^{157}\) https://docs.python.org/3.5/library/constants.html#None

\(^{158}\) https://docs.python.org/3.5/library/functions.html#int

\(^{159}\) https://docs.python.org/3.5/library/stdtypes.html#str

\(^{160}\) https://docs.python.org/3.5/library/constants.html#None
• **queue_len** (*int*) – The length of the queue used to store values read from the circuit. This defaults to 5.

• **charge_time_limit** (*float*) – If the capacitor in the circuit takes longer than this length of time to charge, it is assumed to be dark. The default (0.01 seconds) is appropriate for a 1µF capacitor coupled with the LDR from the CamJam #2 EduKit[163]. You may need to adjust this value for different valued capacitors or LDRs.

• **threshold** (*float*) – Defaults to 0.1. When the average of all values in the internal queue rises above this value, the area will be considered “light”, and all appropriate events will be fired.

• **partial** (*bool*) – When False (the default), the object will not return a value for **is_active** (page 106) 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 198) or **None**) – See **API - Pins** (page 195) for more information (this is an advanced feature which most users can ignore).

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

**Parameters**  
**timeout** (*float* or **None**) – 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* or **None**) – 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 **value** (page 107) currently exceeds **threshold** (page 107) and **False** otherwise.

**pin**  
The **Pin** (page 199) that the device is connected to. This will be **None** if the device has been closed (see the **close()** (page 175) 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 between 0 (dark) and 1 (light).

**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.

---

[161](https://docs.python.org/3.5/library/functions.html#int)  
[162](https://docs.python.org/3.5/library/functions.html#float)  
[163](http://camjam.me/?page_id=623)  
[164](https://docs.python.org/3.5/library/functions.html#float)  
[165](https://docs.python.org/3.5/library/functions.html#bool)  
[166](https://docs.python.org/3.5/library/constants.html#False)  
[167](https://docs.python.org/3.5/library/constants.html#True)  
[168](https://docs.python.org/3.5/library/constants.html#None)  
[169](https://docs.python.org/3.5/library/functions.html#float)  
[170](https://docs.python.org/3.5/library/constants.html#None)  
[171](https://docs.python.org/3.5/library/constants.html#None)  
[172](https://docs.python.org/3.5/library/constants.html#None)  
[173](https://docs.python.org/3.5/library/constants.html#None)  
[174](https://docs.python.org/3.5/library/constants.html#None)  
[175](https://docs.python.org/3.5/library/constants.html#True)  
[176](https://docs.python.org/3.5/library/constants.html#False)  
[177](https://docs.python.org/3.5/library/constants.html#None)
Set this property to None\(^{178}\) (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\(^{179}\) (the default) to disable the event.

### 13.1.5 DistanceSensor (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 105) and represents an HC-SR04 ultrasonic distance sensor, as found in the CamJam #3 EduKit\(^{180}\).

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 330\(\Omega\) resistor to the ECHO pin of the sensor.
4. Connect one end of a 470\(\Omega\) 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\(^{181}\).
6. Finally, connect the VCC pin of the sensor to a 5V pin on the Pi.

Alternatively, the 3V3 tolerant HC-SR04P sensor (which does not require a voltage divider) will work with this class.

**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\(^{182}\) 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 330\(\Omega\) resistor, and two of them in series instead of the 470\(\Omega\) 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:

```python
from gpiozero import DistanceSensor
from time import sleep

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

\(^{178}\) https://docs.python.org/3.5/library/constants.html#None

\(^{179}\) https://docs.python.org/3.5/library/constants.html#None

\(^{180}\) http://camjam.me/?page_id=1035

\(^{181}\) https://en.wikipedia.org/wiki/Voltage_divider

\(^{182}\) https://en.wikipedia.org/wiki/Voltage_divider
Note: For improved accuracy, use the pigpio pin driver rather than the default RPi.GPIO driver (pigpio uses DMA sampling for much more precise edge timing). This is particularly relevant if you’re using Pi 1 or Pi Zero. See Changing the pin factory (page 196) for further information.

Parameters

- **echo** (`int` or `str`) – The GPIO pin which the ECHO pin is connected to. See Pin Numbering (page 3) for valid pin numbers. If this is `None` a `GPIODeviceError` (page 213) will be raised.

- **trigger** (`int` or `str`) – The GPIO pin which the TRIG pin is connected to. See Pin Numbering (page 3) for valid pin numbers. If this is `None` a `GPIODeviceError` (page 213) will be raised.

- **queue_len** (`int`) – The length of the queue used to store values read from the sensor. This defaults to 30.

- **max_distance** (`float`) – The `value` (page 103) attribute reports a normalized value between 0 (too close to measure) and 1 (maximum distance). This parameter specifies the maximum distance expected in meters. This defaults to 1.

- **threshold_distance** (`float`) – Defaults to 0.3. This is the distance (in meters) that will trigger the `in_range` and `out_of_range` events when crossed.

- **partial** (`bool`) – When `False` (the default), the object will not return a value for `is_active` (page 106) until the internal queue has filled with values. Only set this to `True` if you require values immediately after object construction.

- **pin_factory** (`Factory` or `None`) – See API - Pins (page 195) 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` or `None`) – 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` or `None`) – Number of seconds to wait before proceeding. If this is `None` (the default), then wait indefinitely until the device is active.

**distance**

Returns the current distance measured by the sensor in meters. Note that this property will have a value between 0 and `max_distance` (page 103).

---

183 https://docs.python.org/3.5/library/functions.html#int
184 https://docs.python.org/3.5/library/stdtypes.html#str
185 https://docs.python.org/3.5/library/constants.html#None
186 https://docs.python.org/3.5/library/functions.html#int
187 https://docs.python.org/3.5/library/stdtypes.html#str
188 https://docs.python.org/3.5/library/constants.html#None
189 https://docs.python.org/3.5/library/functions.html#int
190 https://docs.python.org/3.5/library/functions.html#float
191 https://docs.python.org/3.5/library/functions.html#float
192 https://docs.python.org/3.5/library/functions.html#bool
193 https://docs.python.org/3.5/library/constants.html#True
194 https://docs.python.org/3.5/library/constants.html#False
195 https://docs.python.org/3.5/library/constants.html#None
196 https://docs.python.org/3.5/library/functions.html#int
197 https://docs.python.org/3.5/library/constants.html#None
198 https://docs.python.org/3.5/library/functions.html#float
199 https://docs.python.org/3.5/library/constants.html#None
200 https://docs.python.org/3.5/library/constants.html#None
201 https://docs.python.org/3.5/library/constants.html#None
```markdown
**echo**
Returns the *Pin* (page 199) that the sensor’s echo is connected to. This is simply an alias for the usual *pin* (page 109) 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* (page 107) attribute. When *distance* (page 102) is equal to *max_distance* (page 103), *value* (page 107) will be 1.

**threshold_distance**
The distance, measured in meters, that will trigger the *when_in_range* (page 103) and *when_out_of_range* (page 103) events when crossed. This is simply a meter-scaled variant of the usual *threshold* (page 107) attribute.

**trigger**
Returns the *Pin* (page 199) that the sensor’s trigger is connected to.

**value**
Returns a value between 0, indicating the reflector is either touching the sensor or is sufficiently near that the sensor can’t tell the difference, and 1, indicating the reflector is at or beyond the specified *max_distance*.

**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*\(^{202}\) (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*\(^{203}\) (the default) to disable the event.

## 13.2 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):

---

\(^{202}\) [https://docs.python.org/3.5/library/constants.html#None](https://docs.python.org/3.5/library/constants.html#None)

\(^{203}\) [https://docs.python.org/3.5/library/constants.html#None](https://docs.python.org/3.5/library/constants.html#None)
The following sections document these base classes for advanced users that wish to construct classes for their own devices.

13.2.1 DigitalInputDevice

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

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

This class extends InputDevice (page 107) with machinery to fire the active and inactive events for devices that operate in a typical digital manner: straightforward on/off states with (reasonably) clean transitions between the two.

Parameters

- **pin** (`int` or `str`) – The GPIO pin that the device is connected to. See Pin Numbering (page 3) for valid pin numbers. If this is `None` a GPIODeviceError (page 213) will be raised.
- **pull_up** (`bool` or `None`) – See description under InputDevice (page 107) for more information.
- **active_state** (`bool` or `None`) – See description under InputDevice (page 107) for more information.
- **bounce_time** (`float` or `None`) – 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 198) or None) – See API - Pins (page 195) for more information (this is an advanced feature which most users can ignore).

**wait_for_active**(timeout=None)

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

**Parameters**

*timeout*(float or None) – 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 or None) – 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.

**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.

**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.

### 13.2.2 SmoothedInputDevice

**class** gpiozero.SmoothedInputDevice(pin, *, pull_up=False, active_state=None, 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 107) 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 106) property.

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

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

**Parameters**

- **pin** (int or str) – The GPIO pin that the device is connected to. See Pin Numbering (page 3) for valid pin numbers. If this is `None` a GPIODeviceError (page 213) will be raised.

- **pull_up** (bool or `None`) – See description under InputDevice (page 107) for more information.

- **active_state** (bool or `None`) – See description under InputDevice (page 107) for more information.

- **threshold** (float) – The value above which the device will be considered “on”.

- **queue_len** (int) – The length of the internal queue which is filled by the background thread.

- **sample_wait** (float) – The length of time to wait between retrieving the state of the underlying device. Defaults to 0.0 indicating that values are retrieved as fast as possible.

- **partial** (bool) – If `False` (the default), attempts to read the state of the device (from the `is_active` (page 106) 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()` which is a good selection for discarding outliers from jittery sensors. The function specified must accept a sequence of numbers and return a single number.

- **ignore** (frozenset or `None`) – The set of values which the queue should ignore, if returned from querying the device’s value.

- **pin_factory** (Factory (page 198) or `None`) – See API - Pins (page 195) for more information (this is an advanced feature which most users can ignore).
**is_active**

Returns `True` \(^{242}\) if the `value` (page 107) currently exceeds `threshold` (page 107) and `False` \(^{243}\) otherwise.

**partial**

If `False` \(^{244}\) (the default), attempts to read the `value` (page 107) or `is_active` (page 106) 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 107) exceeds this amount, then `is_active` (page 106) will return `True` \(^{245}\).

**value**

Returns the average of the values in the internal queue. This is compared to `threshold` (page 107) to determine whether `is_active` (page 106) is `True` \(^{246}\).

### 13.2.3 InputDevice

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

Represents a generic GPIO input device.

This class extends `GPIODevice` (page 108) 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` (page 108) property is adjusted accordingly so that `True` \(^{247}\) still means active regardless of the `pull_up` setting.

**Parameters**

- **pin** (`int` \(^{248}\) or `str` \(^{249}\)) – The GPIO pin that the device is connected to. See `Pin Numbering` (page 3) for valid pin numbers. If this is `None` \(^{250}\) a `GPIODeviceError` (page 213) will be raised.

- **pull_up** (`bool` \(^{251}\) or `None` \(^{252}\)) – If `True` \(^{253}\), the pin will be pulled high with an internal resistor. If `False` \(^{254}\) (the default), the pin will be pulled low. If `None` \(^{255}\), the pin will be floating. As gpiozero cannot automatically guess the active state when not pulling the pin, the `active_state` parameter must be passed.

- **active_state** (`bool` \(^{256}\) or `None` \(^{257}\)) – If `True` \(^{258}\), when the hardware pin state is `HIGH`, the software pin is `HIGH`. If `False` \(^{259}\), the input polarity is reversed: when the hardware pin state is `HIGH`, the software pin state is `LOW`. Use this parameter to set the active state of the underlying pin when configuring it as not pulled (when `pull_up` is...
None). When pull_up is True or False, the active state is automatically set to the proper value.

* pin_factory (Factory (page 198) or None) — See API - Pins (page 195) for more information (this is an advanced feature which most users can ignore).

is_active
Returns True if the device is currently active and False otherwise. This property is usually derived from value (page 108). Unlike value (page 108), this is always a boolean.

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.

13.2.4 GPIODevice

class gpiozero.GPIODevice (pin, pin_factory=None)
Extends Device (page 175). 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 109)).

Parameters pin (int or str) — The GPIO pin that the device is connected to. See Pin Numbering (page 3) for valid pin numbers. If this is None a GPIODeviceError (page 213) will be raised. If the pin is already in use by another device, GPIOPinInUse (page 213) 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:

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

References:

- https://docs.python.org/3.5/library/constants.html#None
- https://docs.python.org/3.5/library/constants.html#True
- https://docs.python.org/3.5/library/constants.html#False
- https://docs.python.org/3.5/library/constants.html#None
- https://docs.python.org/3.5/library/constants.html#str
- https://docs.python.org/3.5/library/functions.html#int
- https://docs.python.org/3.5/library/stdtypes.html#str
- https://docs.python.org/3.5/library/constants.html#None

Chapter 13. API - Input Devices
Device (page 175) descendents can also be used as context managers using the with statement. For example:

```python
>>> 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 108) 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 199) that the device is connected to. This will be None if the device has been closed (see the close (page 175) 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.

---

270 https://docs.python.org/3.5/reference/compound_stmts.html#with
271 https://docs.python.org/3.5/library/const.html#True
272 https://docs.python.org/3.5/library/const.html#None
CHAPTER 14

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 by default. See the Pin Numbering (page 3) section for more information.

14.1 Regular Classes

The following classes are intended for general use with the devices they represent. All classes in this section are concrete (not abstract).

14.1.1 LED

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

Extends DigitalOutputDevice (page 127) 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:

```python
from gpiozero import LED
led = LED(17)
led.on()
```

Parameters

- **pin** (int\(^{273}\) or str\(^{274}\)) – The GPIO pin which the LED is connected to. See Pin Numbering (page 3) for valid pin numbers. If this is None\(^{275}\) a GPIODeviceError

\(^{273}\) https://docs.python.org/3.5/library/functions.html#int
\(^{274}\) https://docs.python.org/3.5/library/stdtypes.html#str
\(^{275}\) https://docs.python.org/3.5/library/constants.html#None
(page 213) will be raised.

- **active_high** (bool\(^{276}\)) – If True\(^{277}\) (the default), the LED will operate normally with the circuit described above. If False\(^{278}\) you should wire the cathode to the GPIO pin, and the anode to a 3V3 pin (via a limiting resistor).

- **initial_value** (bool\(^{279}\) or None\(^{280}\)) – If False\(^{281}\) (the default), the LED will be off initially. If None\(^{282}\), the LED will be left in whatever state the pin is found in when configured for output (warning: this can be on). If True\(^{283}\), the LED will be switched on initially.

- **pin_factory** (Factory (page 198) or None\(^{284}\)) – See API - Pins (page 195) 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\(^{285}\)) – Number of seconds on. Defaults to 1 second.

- **off_time** (float\(^{286}\)) – Number of seconds off. Defaults to 1 second.

- **n** (int\(^{287}\) or None\(^{288}\)) – Number of times to blink; None\(^{289}\) (the default) means forever.

- **background** (bool\(^{290}\)) – If True\(^{291}\) (the default), start a background thread to continue blinking and return immediately. If False\(^{292}\), 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\(^{293}\) if the device is currently active and False\(^{294}\) otherwise. This property is usually derived from value (page 112). Unlike value (page 112), this is always a boolean.

**pin**

The Pin (page 199) that the device is connected to. This will be None\(^{295}\) if the device has been closed (see the close() (page 175) 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 1 if the device is currently active and 0 otherwise. Setting this property changes the state of
the device.

14.1.2 PWMLED
class gpiozero.PWMLED (pin, *, active_high=True, initial_value=0, frequency=100,
pin_factory=None)
Extends PWMOutputDevice (page 128) and represents a light emitting diode (LED) with variable bright-
ness.
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 (int296 or str297) – The GPIO pin which the LED is connected to. See Pin
Numbering (page 3) for valid pin numbers. If this is None298 a GPIODeviceError
(page 213) will be raised.
• active_high (bool299) – If True300 (the default), the on() (page 114) method
will set the GPIO to HIGH. If False301, the on() (page 114) method will set the
GPIO to LOW (the off() (page 114) method always does the opposite).
• initial_value (float302) – 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
None303 cannot be specified (unlike the parent class) as there is no way to tell PWM
not to alter the state of the pin.
• frequency (int304) – The frequency (in Hz) of pulses emitted to drive the LED.
Defaults to 100Hz.
• pin_factory (Factory (page 198) or None305) – See API - Pins (page 195) 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 (float306) – Number of seconds on. Defaults to 1 second.
• off_time (float307) – Number of seconds off. Defaults to 1 second.
• fade_in_time (float308) – Number of seconds to spend fading in. Defaults to 0.
• fade_out_time (float309) – Number of seconds to spend fading out. Defaults to 0.

296 https://docs.python.org/3.5/library/functions.html#int
297 https://docs.python.org/3.5/library/stdtypes.html#str
298 https://docs.python.org/3.5/library/constants.html#None
299 https://docs.python.org/3.5/library/functions.html#bool
300 https://docs.python.org/3.5/library/constants.html#True
301 https://docs.python.org/3.5/library/constants.html#False
302 https://docs.python.org/3.5/library/functions.html#float
303 https://docs.python.org/3.5/library/constants.html#None
304 https://docs.python.org/3.5/library/functions.html#int
305 https://docs.python.org/3.5/library/constants.html#None
306 https://docs.python.org/3.5/library/functions.html#float
307 https://docs.python.org/3.5/library/functions.html#float
308 https://docs.python.org/3.5/library/functions.html#float
309 https://docs.python.org/3.5/library/functions.html#float
• \( n (\text{int}^{310} \text{ or } \text{None}^{311}) \) – Number of times to blink; \( \text{None}^{312} \) (the default) means forever.

• \( \text{background} (\text{bool}^{313}) \) – If \( \text{True}^{314} \) (the default), start a background thread to continue blinking and return immediately. If \( \text{False}^{315} \), only return when the blink is finished (warning: the default value of \( n \) will result in this method never returning).

\[ \text{off}() \]
Turns the device off.

\[ \text{on}() \]
Turns the device on.

\[ \text{pulse}(\text{fade_in_time}=1, \text{fade_out_time}=1, n=\text{None}, \text{background}=\text{True}) \]
Make the device fade in and out repeatedly.

Parameters

• \( \text{fade_in_time} (\text{float}^{316}) \) – Number of seconds to spend fading in. Defaults to 1.

• \( \text{fade_out_time} (\text{float}^{317}) \) – Number of seconds to spend fading out. Defaults to 1.

• \( n (\text{int}^{318} \text{ or } \text{None}^{319}) \) – Number of times to pulse; \( \text{None}^{320} \) (the default) means forever.

• \( \text{background} (\text{bool}^{321}) \) – If \( \text{True}^{322} \) (the default), start a background thread to continue pulsing and return immediately. If \( \text{False}^{323} \), only return when the pulse is finished (warning: the default value of \( n \) will result in this method never returning).

\[ \text{toggle}() \]
Toggle the state of the device. If the device is currently off (\( \text{value} \) (page 114) is 0.0), this changes it to “fully” on (\( \text{value} \) (page 114) is 1.0). If the device has a duty cycle (\( \text{value} \) (page 114)) of 0.1, this will toggle it to 0.9, and so on.

\[ \text{is_lit} \]
Returns \( \text{True}^{324} \) if the device is currently active (\( \text{value} \) (page 114) is non-zero) and \( \text{False}^{325} \) otherwise.

\[ \text{pin} \]
The \( \text{Pin} \) (page 199) that the device is connected to. This will be \( \text{None}^{326} \) if the device has been closed (see the \( \text{close}() \) (page 175) method). When dealing with GPIO pins, query \( \text{pin.number} \) to discover the GPIO pin (in BCM numbering) that the device is connected to.

\[ \text{value} \]
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.

\[ ^{310} \text{https://docs.python.org/3.5/library/functions.html#int} \]
\[ ^{311} \text{https://docs.python.org/3.5/library/constants.html#None} \]
\[ ^{312} \text{https://docs.python.org/3.5/library/constants.html#None} \]
\[ ^{313} \text{https://docs.python.org/3.5/library/functions.html#bool} \]
\[ ^{314} \text{https://docs.python.org/3.5/library/constants.html#True} \]
\[ ^{315} \text{https://docs.python.org/3.5/library/constants.html#False} \]
\[ ^{316} \text{https://docs.python.org/3.5/library/functions.html#float} \]
\[ ^{317} \text{https://docs.python.org/3.5/library/functions.html#float} \]
\[ ^{318} \text{https://docs.python.org/3.5/library/functions.html#int} \]
\[ ^{319} \text{https://docs.python.org/3.5/library/constants.html#None} \]
\[ ^{320} \text{https://docs.python.org/3.5/library/constants.html#None} \]
\[ ^{321} \text{https://docs.python.org/3.5/library/functions.html#bool} \]
\[ ^{322} \text{https://docs.python.org/3.5/library/constants.html#True} \]
\[ ^{323} \text{https://docs.python.org/3.5/library/constants.html#False} \]
\[ ^{324} \text{https://docs.python.org/3.5/library/constants.html#True} \]
\[ ^{325} \text{https://docs.python.org/3.5/library/constants.html#False} \]
\[ ^{326} \text{https://docs.python.org/3.5/library/constants.html#None} \]
14.1.3 RGBLED

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

Extends Device (page 175) 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 should use three limiting resistors (one per anode).

The following code will make the LED yellow:

```python
from gpiozero import RGBLED
led = RGBLED(2, 3, 4)
led.color = (1, 1, 0)
```

The colorzero library is also supported:

```python
from gpiozero import RGBLED
from colorzero import Color
led = RGBLED(2, 3, 4)
led.color = Color('yellow')
```

Parameters

- `red (int328 or str329) – The GPIO pin that controls the red component of the RGB LED. See Pin Numbering (page 3) for valid pin numbers. If this is None330 a GPIODeviceError (page 213) will be raised.
- `green (int331 or str332) – The GPIO pin that controls the green component of the RGB LED.
- `blue (int333 or str334) – The GPIO pin that controls the blue component of the RGB LED.
- `active_high (bool335) – Set to True336 (the default) for common cathode RGB LEDs. If you are using a common anode RGB LED, set this to False337.
- `initial_value (Color338 or tuple339) – The initial color for the RGB LED. Defaults to black (0, 0, 0).
- `pwm (bool340) – If True341 (the default), construct PWMLED (page 113) instances for each component of the RGBLED. If False342, construct regular LED (page 111) instances, which prevents smooth color graduations.
- `pin_factory (Factory (page 198) or None343) – See API - Pins (page 195) for more information (this is an advanced feature which most users can ignore).
**Gpiozero Documentation, Release 1.5.1**

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

Make the device turn on and off repeatedly.

**Parameters**

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

**off()**

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

**on()**

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

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

Make the device fade in and out repeatedly.

**Parameters**

- **fade_in_time** (*float*) – Number of seconds to spend fading in. Defaults to 1.
- **fade_out_time** (*float*) – Number of seconds to spend fading out. Defaults to 1.
• **on_color** *(Color or tuple)* – The color to use when the LED is “on”. Defaults to white.

• **off_color** *(Color or tuple)* – The color to use when the LED is “off”. Defaults to black.

• **n** *(int or None)* – Number of times to pulse; None (the default) means forever.

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

**toggle**

Toggles the state of the device. If the device is currently off *(value (page 117) is (0, 0, 0)), this changes it to “fully” on *(value (page 117) is (1, 1, 1)).* If the device has a specific color, this method inverts the color.

**blue**

Represents the blue element of the LED as a Blue object.

**color**

Represents the color of the LED as a Color object.

**green**

Represents the green element of the LED as a Green object.

**is_lit**

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

**red**

Represents the red element of the LED as a Red object.

**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, red would be *(1, 0, 0)* and yellow would be *(1, 1, 0)*, while orange would be *(1, 0.5, 0)*.

### 14.1.4 Buzzer

**class gpiozero.Buzzer** *(pin, *, active_high=True, initial_value=False, pin_factory=None)*  
Extends DigitalOutputDevice (page 127) and represents a digital buzzer component.

**Note:** This interface is only capable of simple on/off commands, and is not capable of playing a variety of tones (see TonalBuzzer (page 119)).

---

365 https://docs.python.org/3.5/library/stdtypes.html#tuple
367 https://docs.python.org/3.5/library/stdtypes.html#tuple
368 https://docs.python.org/3.5/library/functions.html#int
369 https://docs.python.org/3.5/library/constants.html#None
370 https://docs.python.org/3.5/library/constants.html#False
371 https://docs.python.org/3.5/library/functions.html#bool
372 https://docs.python.org/3.5/library/functions.html#int
373 https://docs.python.org/3.5/library/constants.html#True
377 https://docs.python.org/3.5/library/constants.html#True
378 https://docs.python.org/3.5/library/constants.html#False
380 https://docs.python.org/3.5/library/constants.html#True
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:

```python
from gpiozero import Buzzer
bz = Buzzer(3)
bz.on()
```

**Parameters**

- `pin (int or str)` – The GPIO pin which the buzzer is connected to. See *Pin Numbering* (page 3) for valid pin numbers. If this is `None` a `GPIODeviceError` (page 213) will be raised.

- `active_high (bool)` – If `True` (the default), the buzzer will operate normally with the circuit described above. If `False` you should wire the cathode to the GPIO pin, and the anode to a 3V3 pin.

- `initial_value (bool or None)` – 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 or None)` – See *API - Pins* (page 195) for more information (this is an advanced feature which most users can ignore).

```python
beep(on_time=1, off_time=1, n=None, background=True)
```

Make the device turn on and off repeatedly.

**Parameters**

- `on_time (float)` – Number of seconds on. Defaults to 1 second.

- `off_time (float)` – Number of seconds off. Defaults to 1 second.

- `n (int or None)` – Number of times to blink; `None` (the default) means forever.

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

```python
off()
```

Turns the device off.

---

381 [https://docs.python.org/3.5/library/functions.html#int](https://docs.python.org/3.5/library/functions.html#int)
382 [https://docs.python.org/3.5/library/stdtypes.html#str](https://docs.python.org/3.5/library/stdtypes.html#str)
383 [https://docs.python.org/3.5/library/constants.html#None](https://docs.python.org/3.5/library/constants.html#None)
384 [https://docs.python.org/3.5/library/functions.html#bool](https://docs.python.org/3.5/library/functions.html#bool)
385 [https://docs.python.org/3.5/library/constants.html#True](https://docs.python.org/3.5/library/constants.html#True)
386 [https://docs.python.org/3.5/library/constants.html#False](https://docs.python.org/3.5/library/constants.html#False)
387 [https://docs.python.org/3.5/library/functions.html#bool](https://docs.python.org/3.5/library/functions.html#bool)
388 [https://docs.python.org/3.5/library/constants.html#None](https://docs.python.org/3.5/library/constants.html#None)
389 [https://docs.python.org/3.5/library/constants.html#False](https://docs.python.org/3.5/library/constants.html#False)
390 [https://docs.python.org/3.5/library/constants.html#None](https://docs.python.org/3.5/library/constants.html#None)
391 [https://docs.python.org/3.5/library/constants.html#True](https://docs.python.org/3.5/library/constants.html#True)
392 [https://docs.python.org/3.5/library/constants.html#None](https://docs.python.org/3.5/library/constants.html#None)
393 [https://docs.python.org/3.5/library/constants.html#None](https://docs.python.org/3.5/library/constants.html#None)
394 [https://docs.python.org/3.5/library/functions.html#float](https://docs.python.org/3.5/library/functions.html#float)
395 [https://docs.python.org/3.5/library/functions.html#float](https://docs.python.org/3.5/library/functions.html#float)
396 [https://docs.python.org/3.5/library/functions.html#int](https://docs.python.org/3.5/library/functions.html#int)
397 [https://docs.python.org/3.5/library/constants.html#None](https://docs.python.org/3.5/library/constants.html#None)
398 [https://docs.python.org/3.5/library/constants.html#True](https://docs.python.org/3.5/library/constants.html#True)
399 [https://docs.python.org/3.5/library/constants.html#False](https://docs.python.org/3.5/library/constants.html#False)
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\(^{401}\) if the device is currently active and False\(^{402}\) otherwise. This property is usually derived from value (page 119). Unlike value (page 119), this is always a boolean.

pin
The Pin (page 199) that the device is connected to. This will be None\(^{403}\) if the device has been closed (see the close() (page 175) 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 1 if the device is currently active and 0 otherwise. Setting this property changes the state of the device.

14.1.5 TonalBuzzer
class gpiozero.TonalBuzzer(pin, *, initial_value=None, mid_tone=Tone('A4'), octaves=1, pin_factory=None)
Extends CompositeDevice (page 164) and represents a tonal buzzer.

Parameters

- **pin**\(^{404}\) or **str**\(^{405}\) – The GPIO pin which the buzzer is connected to. See Pin Numbering (page 3) for valid pin numbers. If this is None\(^{406}\) a GPIODeviceError (page 213) will be raised.

- **initial_value**\(^{407}\) – If None\(^{408}\) (the default), the buzzer will be off initially. Values between -1 and 1 can be specified as an initial value for the buzzer.

- **mid_tone**\(^{409}\) or **str**\(^{410}\) – The tone which is represented the device’s middle value (0). The default is “A4” (MIDI note 69).

- **octaves**\(^{411}\) – The number of octaves to allow away from the base note. The default is 1, meaning a value of -1 goes one octave below the base note, and one above, i.e. from A3 to A5 with the default base note of A4.

- **pin_factory**\(^{412}\) – See API - Pins (page 195) for more information (this is an advanced feature which most users can ignore).

Note: Note that this class does not currently work with PiGPIOFactory (page 207).

play(*tone*)
Play the given tone. This can either be an instance of Tone (page 187) or can be anything that could be used to construct an instance of Tone (page 187).

For example:

401 https://docs.python.org/3.5/library/constants.html#True
402 https://docs.python.org/3.5/library/constants.html#False
403 https://docs.python.org/3.5/library/constants.html#None
404 https://docs.python.org/3.5/library/functions.html#int
405 https://docs.python.org/3.5/library/stdtypes.html#str
406 https://docs.python.org/3.5/library/functions.html#int
407 https://docs.python.org/3.5/library/functions.html#float
408 https://docs.python.org/3.5/library/constants.html#None
409 https://docs.python.org/3.5/library/functions.html#int
410 https://docs.python.org/3.5/library/stdtypes.html#str
411 https://docs.python.org/3.5/library/functions.html#int
412 https://docs.python.org/3.5/library/constants.html#None
>>> from gpiozero import TonalBuzzer
>>> from gpiozero.tones import Tone

>>> b = TonalBuzzer(17)
>>> b.play(Tone("A4"))
>>> b.play(Tone(220.0))  # Hz
>>> b.play(Tone(60))    # middle C in MIDI notation
>>> b.play("A4")
>>> b.play(220.0)
>>> b.play(60)

def stop()
    """Turn the buzzer off. This is equivalent to setting value (page 120) to None."

    is_active
    """Returns True if the buzzer is currently playing, otherwise False."

max_tone
The highest tone that the buzzer can play, i.e. the tone played when value (page 120) is 1.

mid_tone
The middle tone available, i.e. the tone played when value (page 120) is 0.

min_tone
The lowest tone that the buzzer can play, i.e. the tone played when value (page 120) is -1.

octaves
The number of octaves available (above and below mid_tone).

tone
Returns the Tone (page 187) that the buzzer is currently playing, or None if the buzzer is silent.
This property can also be set to play the specified tone.

value
Represents the state of the buzzer as a value between -1 (representing the minimum tone) and 1 (representing the maximum tone). This can also be the special value None indicating that the buzzer is currently silent.

14.1.6 Motor
class gpiozero.Motor (forward, backward, *, pwm=True, pin_factory=None)
    Extends CompositeDevice (page 164) and represents a generic motor connected to a bi-directional motor driver circuit (i.e. an H-bridge).

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

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

from gpiozero import Motor
motor = Motor(17, 18)
motor.forward()

Parameters

https://docs.python.org/3.5/library/constants.html#None
https://docs.python.org/3.5/library/constants.html#True
https://docs.python.org/3.5/library/constants.html#False
https://docs.python.org/3.5/library/constants.html#None
https://en.wikipedia.org/wiki/H_bridge
https://en.wikipedia.org/wiki/H_bridge
• **forward** (*int* *420* *or str* *421*) – The GPIO pin that the forward input of the motor driver chip is connected to. See *Pin Numbering* (page 3) for valid pin numbers. If this is *None* *422* a GPIODeviceError (page 213) will be raised.

• **backward** (*int* *423* *or str* *424*) – The GPIO pin that the backward input of the motor driver chip is connected to. See *Pin Numbering* (page 3) for valid pin numbers. If this is *None* *425* a GPIODeviceError (page 213) will be raised.

• **enable** (*int* *426* *or str* *427* *or None* *428*) – The GPIO pin that enables the motor. Required for some motor controller boards. See *Pin Numbering* (page 3) for valid pin numbers.

• **pwm** (*bool* *429*) – If *True* (the default), construct PWMOutputDevice (page 128) instances for the motor controller pins, allowing both direction and variable speed control. If *False* *431*, construct DigitalOutputDevice (page 127) instances, allowing only direction control.

• **pin_factory** (*Factory* *422* *or None* *432*) – See *API - Pins* (page 195) for more information (this is an advanced feature which most users can ignore).

```python
backward(speed=1)
```
Drive the motor backwards.

**Parameters**

**speed** (*float* *433*) – 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* *434* when the class was constructed (and only 0 or 1 if not).

```python
forward(speed=1)
```
Drive the motor forwards.

**Parameters**

**speed** (*float* *435*) – 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* *436* when the class was constructed (and only 0 or 1 if not).

```python
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.

```python
stop()
```
Stop the motor.

```python
is_active
```
Returns *True* *437* if the motor is currently running and *False* *438* otherwise.

```python
value
```
Represents the speed of the motor as a floating point value between -1 (full speed backward) and 1 (full speed forward), with 0 representing stopped.
14.1.7 PhaseEnableMotor

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

Extends CompositeDevice (page 164) 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”:

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

Parameters

- **phase** *(int or str)* – The GPIO pin that the phase (direction) input of the
  motor driver chip is connected to. See Pin Numbering (page 3) for valid pin numbers.
  If this is None a GPIODeviceError (page 213) will be raised.

- **enable** *(int or str)* – The GPIO pin that the enable (speed) input of the
  motor driver chip is connected to. See Pin Numbering (page 3) for valid pin numbers.
  If this is None a GPIODeviceError (page 213) will be raised.

- **pwm** *(bool)* – If True (the default), construct PWMOutputDevice (page 128)
  instances for the motor controller pins, allowing both direction and variable speed con-
  trol. If False, construct DigitalOutputDevice (page 127) instances, allowing
  only direction control.

- **pin_factory** *(Factory or None)* – See API - Pins (page 195) for
  more information (this is an advanced feature which most users can ignore).

**backward**(speed=1)

Drive the motor backwards.

**Parameters**

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

**forward**(speed=1)

Drive the motor forwards.

**Parameters**

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

**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.

**is_active**

Returns True if the motor is currently running and False otherwise.
value
Represents the speed of the motor as a floating point value between -1 (full speed backward) and 1 (full speed forward).

14.1.8 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 164) 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:

```python
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)
```

You can also use the `value` (page 124) property to move the servo to a particular position, on a scale from -1 (min) to 1 (max) where 0 is the mid-point:

```python
from gpiozero import Servo

servo = Servo(17)

servo.value = 0.5
```

Parameters
- **pin** (`int` or `str`) – The GPIO pin that the servo is connected to. See Pin Numbering (page 3) for valid pin numbers. If this is `None` a GPIODeviceError (page 213) will be raised.
- **initial_value** (`float`) – If 0 (the default), the device’s mid-point will be set initially. Other values between -1 and +1 can be specified as an initial position. `None` means to start the servo un-controlled (see `value` (page 124)).
- **min_pulse_width** (`float`) – The pulse width corresponding to the servo’s minimum position. This defaults to 1ms.
- **max_pulse_width** (`float`) – The pulse width corresponding to the servo’s maximum position. This defaults to 2ms.

---

453 https://docs.python.org/3.5/library/functions.html#int
454 https://docs.python.org/3.5/library/stdtypes.html#str
455 https://docs.python.org/3.5/library/constants.html#None
456 https://docs.python.org/3.5/library/constants.html#float
457 https://docs.python.org/3.5/library/constants.html#float
458 https://docs.python.org/3.5/library/constants.html#float
459 https://docs.python.org/3.5/library/constants.html#float

14.1. Regular Classes 123
• **frame_width** (*float*) – The length of time between servo control pulses measured in seconds. This defaults to 20ms which is a common value for servos.

• **pin_factory** (*Factory (page 198) or None*) – See API - Pins (page 195) for more information (this is an advanced feature which most users can ignore).

```python
def detach()
    Temporarily disable control of the servo. This is equivalent to setting value (page 124) to None.
```

```python
def max()
    Set the servo to its maximum position.
```

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

```python
def min()
    Set the servo to its minimum position.
```

```python
def frame_width
    The time between control pulses, measured in seconds.
```

```python
def is_active
    Composite devices are considered “active” if any of their constituent devices have a “truthy” value.
```

```python
def max_pulse_width
    The control pulse width corresponding to the servo’s maximum position, measured in seconds.
```

```python
def min_pulse_width
    The control pulse width corresponding to the servo’s minimum position, measured in seconds.
```

```python
def pulse_width
    Returns the current pulse width controlling the servo.
```

```python
def 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.
```

### 14.1.9 AngularServo

```python
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 123) 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 123) 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:

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

---

460 [https://docs.python.org/3.5/library/functions.html#float](https://docs.python.org/3.5/library/functions.html#float)
461 [https://docs.python.org/3.5/library/constants.html#None](https://docs.python.org/3.5/library/constants.html#None)
462 [https://docs.python.org/3.5/library/constants.html#None](https://docs.python.org/3.5/library/constants.html#None)
463 [https://docs.python.org/3.5/library/constants.html#None](https://docs.python.org/3.5/library/constants.html#None)
You should now be able to construct an AngularServo instance with the correct bounds:

```python
>>> from gpiozero import AngularServo
>>> s = AngularServo(17, min_angle=-42, max_angle=44)
>>> s.angle = 0.0
0.0
>>> s.angle = 15
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`][464] or [`str`][465]) – The GPIO pin that the servo is connected to. See *Pin Numbering* (page 3) for valid pin numbers. If this is `None`[466] a `GPIODeviceError` (page 213) will be raised.

- **initial_angle** ([`float`][467]) – 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`[468] means to start the servo un-controlled (see *value* (page 126)).

- **min_angle** ([`float`][469]) – 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`][470]) – 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`][471]) – The pulse width corresponding to the servo’s minimum position. This defaults to 1ms.

- **max_pulse_width** ([`float`][472]) – The pulse width corresponding to the servo’s maximum position. This defaults to 2ms.

- **frame_width** ([`float`][473]) – 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`][474] or `None`) – See *API - Pins* (page 195) for more information (this is an advanced feature which most users can ignore).

```python
max()
Set the servo to its maximum position.

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

---

464 https://docs.python.org/3.5/library/functions.html#int
465 https://docs.python.org/3.5/library/stdtypes.html#str
466 https://docs.python.org/3.5/library/constants.html#None
467 https://docs.python.org/3.5/library/functions.html#float
468 https://docs.python.org/3.5/library/constants.html#None
469 https://docs.python.org/3.5/library/functions.html#float
470 https://docs.python.org/3.5/library/functions.html#float
471 https://docs.python.org/3.5/library/functions.html#float
472 https://docs.python.org/3.5/library/functions.html#float
473 https://docs.python.org/3.5/library/functions.html#float
474 https://docs.python.org/3.5/library/constants.html#None
**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 \texttt{min\_angle} (page 126) and \texttt{max\_angle} (page 126) have been set appropriately in the constructor.

This can also be the special value \texttt{None}\ref{https://docs.python.org/3.5/library/constants.html#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.

**is\_active**
Composite devices are considered “active” if any of their constituent devices have a “truthy” value.

**max\_angle**
The maximum angle that the servo will rotate to when \texttt{max()} (page 125) is called.

**min\_angle**
The minimum angle that the servo will rotate to when \texttt{min()} (page 125) is called.

**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 \texttt{None}\ref{https://docs.python.org/3.5/library/constants.html#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.

### 14.2 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):

\footnote{https://docs.python.org/3.5/library/constants.html#None}
\footnote{https://docs.python.org/3.5/library/constants.html#None}
The following sections document these base classes for advanced users that wish to construct classes for their own devices.

14.2.1 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 130) with a blink() (page 127) method which uses an optional background thread to handle toggling the device state without further interaction.

Parameters

- **pin (int\(^{477}\) or str\(^{478}\))** – The GPIO pin that the device is connected to. See Pin Numbering (page 3) for valid pin numbers. If this is None\(^{479}\) a GPIODeviceError (page 213) will be raised.
- **active_high (bool\(^{480}\))** – If True\(^{481}\) (the default), the on() (page 128) method will set the GPIO to HIGH. If False\(^{482}\), the on() (page 128) method will set the GPIO to LOW (the off() (page 127) method always does the opposite).
- **initial_value (bool\(^{483}\) or None\(^{484}\))** – If False\(^{485}\) (the default), the device will be off initially. If None\(^{486}\), the device will be left in whatever state the pin is found in when configured for output (warning: this can be on). If True\(^{487}\), the device will be switched on initially.
- **pin_factory (Factory (page 198) or None\(^{488}\))** – See API - Pins (page 195) 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\(^{489}\))** – Number of seconds on. Defaults to 1 second.
- **off_time (float\(^{490}\))** – Number of seconds off. Defaults to 1 second.
- **n (int\(^{491}\) or None\(^{492}\))** – Number of times to blink; None\(^{493}\) (the default) means forever.
- **background (bool\(^{494}\))** – If True\(^{495}\) (the default), start a background thread to continue blinking and return immediately. If False\(^{496}\), only return when the blink is finished (warning: the default value of n will result in this method never returning).

\(^{477}\)https://docs.python.org/3.5/library/functions.html#int  
^{478}\)https://docs.python.org/3.5/library/stdtypes.html#str  
^{479}\)https://docs.python.org/3.5/library/constants.html#None  
^{480}\)https://docs.python.org/3.5/library/functions.html#bool  
^{481}\)https://docs.python.org/3.5/library/constants.html#True  
^{482}\)https://docs.python.org/3.5/library/constants.html#False  
^{483}\)https://docs.python.org/3.5/library/functions.html#bool  
^{484}\)https://docs.python.org/3.5/library/constants.html#None  
^{485}\)https://docs.python.org/3.5/library/constants.html#False  
^{486}\)https://docs.python.org/3.5/library/constants.html#None  
^{487}\)https://docs.python.org/3.5/library/constants.html#True  
^{488}\)https://docs.python.org/3.5/library/constants.html#None  
^{489}\)https://docs.python.org/3.5/library/functions.html#float  
^{490}\)https://docs.python.org/3.5/library/functions.html#float  
^{491}\)https://docs.python.org/3.5/library/functions.html#int  
^{492}\)https://docs.python.org/3.5/library/functions.html#int  
^{493}\)https://docs.python.org/3.5/library/constants.html#None  
^{494}\)https://docs.python.org/3.5/library/constants.html#None  
^{495}\)https://docs.python.org/3.5/library/constants.html#True  
^{496}\)https://docs.python.org/3.5/library/constants.html#False
off()

Turns the device off.

on()

Turns the device on.

value

Returns 1 if the device is currently active and 0 otherwise. Setting this property changes the state of the device.

14.2.2 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 or str) – The GPIO pin that the device is connected to. See Pin Numbering (page 3) for valid pin numbers. If this is None a GPIODeviceError (page 213) will be raised.

• active_high (bool) – If True (the default), the on() (page 129) method will set the GPIO to HIGH. If False, the on() (page 129) method will set the GPIO to LOW (the off() (page 129) method always does the opposite).

• initial_value (float) – If 0 (the default), the device’s duty cycle will be 0 initially. Other values between 0 and 1 can be specified as an initial duty cycle. Note that None cannot be specified (unlike the parent class) as there is no way to tell PWM not to alter the state of the pin.

• frequency (int) – The frequency (in Hz) of pulses emitted to drive the device. Defaults to 100Hz.

• pin_factory (Factory or None) – See API - Pins (page 195) 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) – Number of seconds on. Defaults to 1 second.

• off_time (float) – Number of seconds off. Defaults to 1 second.

• fade_in_time (float) – Number of seconds to spend fading in. Defaults to 0.

• fade_out_time (float) – Number of seconds to spend fading out. Defaults to 0.

497 https://docs.python.org/3.5/library/functions.html#int
498 https://docs.python.org/3.5/library/stdtypes.html#str
499 https://docs.python.org/3.5/library/constants.html#None
500 https://docs.python.org/3.5/library/functions.html#bool
501 https://docs.python.org/3.5/library/constants.html#True
502 https://docs.python.org/3.5/library/constants.html#False
503 https://docs.python.org/3.5/library/functions.html#float
504 https://docs.python.org/3.5/library/constants.html#None
505 https://docs.python.org/3.5/library/functions.html#int
506 https://docs.python.org/3.5/library/constants.html#float
• **n** *(int or None)* – Number of times to blink; None (the default) means forever.

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

off()

  Turns the device off.

on()

  Turns the device on.

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

  Make the device fade in and out repeatedly.

Parameters

• **fade_in_time** *(float)* – Number of seconds to spend fading in. Defaults to 1.

• **fade_out_time** *(float)* – Number of seconds to spend fading out. Defaults to 1.

• **n** *(int or None)* – Number of times to pulse; None (the default) means forever.

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

toggle()

  Toggle the state of the device. If the device is currently off (value (page 129) is 0.0), this changes it to “fully” on (value (page 129) is 1.0). If the device has a duty cycle (value (page 129)) 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 129) 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.
14.2.3 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 108) to add facilities common to GPIO output devices: an on() (page 130) method to switch the device on, a corresponding off() (page 130) method, and a toggle() (page 130) method.

Parameters

• pin (int or str) – The GPIO pin that the device is connected to. See Pin Numbering (page 3) for valid pin numbers. If this is None a GPIODeviceError (page 213) will be raised.

• active_high (bool) – If True (the default), the on() (page 130) method will set the GPIO to HIGH. If False, the on() (page 130) method will set the GPIO to LOW (the off() (page 130) method always does the opposite).

• initial_value (bool or None) – 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 or None) – See API - Pins (page 195) for more information (this is an advanced feature which most users can ignore).

off()  
Turns the device off.

on()  
Turns the device on.

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

active_high  
When True, the value (page 130) property is True when the device’s pin (page 109) is high. When False, the value (page 130) 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 130) (i.e. changing this property doesn’t change the device’s pin state - it just changes how that state is interpreted).

value  
Returns 1 if the device is currently active and 0 otherwise. Setting this property changes the state of the device.

527 https://docs.python.org/3.5/library/functions.html#int
528 https://docs.python.org/3.5/library/stdtypes.html#str
529 https://docs.python.org/3.5/library/constants.html#None
530 https://docs.python.org/3.5/library/functions.html#bool
531 https://docs.python.org/3.5/library/constants.html#True
532 https://docs.python.org/3.5/library/constants.html#False
533 https://docs.python.org/3.5/library/functions.html#bool
534 https://docs.python.org/3.5/library/constants.html#None
535 https://docs.python.org/3.5/library/constants.html#False
536 https://docs.python.org/3.5/library/constants.html#None
537 https://docs.python.org/3.5/library/constants.html#True
538 https://docs.python.org/3.5/library/constants.html#None
539 https://docs.python.org/3.5/library/constants.html#False
540 https://docs.python.org/3.5/library/constants.html#None
541 https://docs.python.org/3.5/library/constants.html#None
542 https://docs.python.org/3.5/library/constants.html#False
14.2.4 GPIODevice

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

Extends Device (page 175). 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 109)).

Parameters pin (int\(^{543}\) or str\(^{544}\)) – The GPIO pin that the device is connected to. See
Pin Numbering (page 3) for valid pin numbers. If this is None\(^{545}\) a GPIODeviceError
(page 213) will be raised. If the pin is already in use by another device, GPIOPinInUse
(page 213) 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:

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

Device (page 175) descendents can also be used as context managers using the with\(^{546}\) statement.
For example:

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

closed

Returns True\(^{547}\) if the device is closed (see the close() (page 108) 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 199) that the device is connected to. This will be None\(^{548}\) if the device has been
closed (see the close() (page 175) 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.

---

543 https://docs.python.org/3.5/library/functions.html#int
544 https://docs.python.org/3.5/library/stdtypes.html#str
545 https://docs.python.org/3.5/library/constants.html#None
546 https://docs.python.org/3.5/reference/compound_stmts.html#with
547 https://docs.python.org/3.5/library/constants.html#True
548 https://docs.python.org/3.5/library/constants.html#None

14.2. Base Classes
SPI stands for **Serial Peripheral Interface**[^wikipedia] 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.

### 15.1 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 213) being raised. However, you can omit any arguments from either scheme. The defaults are:

• *port* and *device* both default to 0.

• *clock_pin* defaults to 11, *mosi_pin* defaults to 10, *miso_pin* defaults to 9, and *select_pin* defaults to 8.

• As with other GPIO based devices you can optionally specify a *pin_factory* argument overriding the default pin factory (see *API - Pins* (page 195) for more information).

Hence the following constructors are all equivalent:

```python
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.

## 15.2 Analog to Digital Converters (ADC)

The following classes are intended for general use with the integrated circuits they are named after. All classes in this section are concrete (not abstract).

### 15.2.1 MCP3001

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

The MCP3001\(^{550}\) 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).

### 15.2.2 MCP3002

**class** `gpiozero.MCP3002(channel=0, differential=False, max_voltage=3.3, **spi_args)`

The MCP3002\(^{551}\) 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 135) 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).


15.2.3 MCP3004

```python
class gpiozero.MCP3004(channel=0, differential=False, max_voltage=3.3, **spi_args)
```

The MCP3004\(^{552}\) 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.

**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 135) 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).

15.2.4 MCP3008

```python
class gpiozero.MCP3008(channel=0, differential=False, max_voltage=3.3, **spi_args)
```

The MCP3008\(^{553}\) 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 135) 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).

15.2.5 MCP3201

```python
class gpiozero.MCP3201(max_voltage=3.3, **spi_args)
```

The MCP3201\(^{554}\) 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).

\(^{552}\) http://www.farnell.com/datasheets/808965.pdf
\(^{553}\) http://www.farnell.com/datasheets/808965.pdf
\(^{554}\) http://www.farnell.com/datasheets/1669366.pdf
15.2.6 MCP3202

class gpiozero.MCP3202 (channel=0, differential=False, max_voltage=3.3, **spi_args)

The MCP3202\(^555\) 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 135) 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).

15.2.7 MCP3204

class gpiozero.MCP3204 (channel=0, differential=False, max_voltage=3.3, **spi_args)

The MCP3204\(^556\) is a 12-bit analog to digital converter with 4 channels (0-3).

channel

The channel to read data from. The MCP3008/3208/3304 have 8 channels (0-7), while the MCP3004/3204/3302 have 4 channels (0-3), 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 135) 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).

15.2.8 MCP3208

class gpiozero.MCP3208 (channel=0, differential=False, max_voltage=3.3, **spi_args)

The MCP3208\(^557\) 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).

\(^{555}\) http://www.farnell.com/datasheets/1669376.pdf
\(^{556}\) http://www.farnell.com/datasheets/808967.pdf
\(^{557}\) http://www.farnell.com/datasheets/808967.pdf
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 135) 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).

### 15.2.9 MCP3301

**class** `gpiozero.MCP3301(max_voltage=3.3, **spi_args)`

The MCP3301\(^{558}\) 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).

### 15.2.10 MCP3302

**class** `gpiozero.MCP3302(channel=0, differential=False, max_voltage=3.3, **spi_args)`

The MCP3302\(^{559}\) 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).

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 137) 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).

### 15.2.11 MCP3304

**class** `gpiozero.MCP3304(channel=0, differential=False, max_voltage=3.3, **spi_args)`

The MCP3304\(^{560}\) 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.

---

\(^{558}\) [http://www.farnell.com/datasheets/1669397.pdf]

\(^{559}\) [http://www.farnell.com/datasheets/1486116.pdf]

\(^{560}\) [http://www.farnell.com/datasheets/1486116.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 MCP3304 (page 137) 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).

15.3 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):

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

```python
from gpiozero import MCP3008
pot = MCP3008(0)
print(pot.value)
```

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

```python
from gpiozero import MCP3008, PWMLED
pot = MCP3008(0)
led = PWMLED(17)
led.source = pot
```

The `voltage` 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.

15.3.2 SPIDevice

```python
from gpiozero import MCP3008, PWMLED
pot = MCP3008(0)
led = PWMLED(17)
led.source = pot
```

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

```python
from gpiozero import MCP3008
pot = MCP3008(0)
pot.value
```

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

```python
from gpiozero import MCP3008, PWMLED
pot = MCP3008(0)
led = PWMLED(17)
led.source = pot
```

The `voltage` 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.
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:

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

Device (page 175) descendents can also be used as context managers using the with statement. For example:

```python
>>> 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 139) method). Once a device is closed you can no longer use any other methods or properties to control or query the device.

562 https://docs.python.org/3.5/reference/compound_stmts.html#with
563 https://docs.python.org/3.5/library/constants.html#True
CHAPTER 16

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 93) and API - Output Devices (page 111) provided by GPIO Zero. See those pages for more information on using components individually.

**Note:** All GPIO pin numbers use Broadcom (BCM) numbering by default. See the Pin Numbering (page 3) section for more information.

16.1 Regular Classes

The following classes are intended for general use with the devices they are named after. All classes in this section are concrete (not abstract).

16.1.1 LEDBoard

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

Extends LEDCollection (page 164) 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:

```python
from gpiozero import LEDBoard

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

**Parameters**

- **pins** – Specify the GPIO pins that the LEDs of the board are attached to. See Pin Numbering (page 3) for valid pin numbers. You can designate as many pins as necessary. You can also specify LEDBoard (page 141) instances to create trees of LEDs.
• **pwm** (bool) – If True, construct **PWMLED** instances for each pin. If False (the default), construct regular **LED** instances.

• **active_high** (bool) – If True (the default), the `on()` method will set all the associated pins to HIGH. If False, the `on()` method will set all pins to LOW (the `off()` method always does the opposite).

• **initial_value** (bool or None) – 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 or None) – See **API - Pins** (page 195) 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 141) instances to create trees of LEDs.

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

Make all the LEDs turn on and off repeatedly.

**Parameters**

• **on_time** (float) – Number of seconds on. Defaults to 1 second.

• **off_time** (float) – Number of seconds off. Defaults to 1 second.

• **fade_in_time** (float) – Number of seconds to spend fading in. Defaults to 0. Must be 0 if `pwm` was `False` when the class was constructed (ValueError will be raised if not).

• **fade_out_time** (float) – Number of seconds to spend fading out. Defaults to 0. Must be 0 if `pwm` was `False` when the class was constructed (ValueError will be raised if not).

• **n** (int or None) – Number of times to blink; None (the default) means forever.

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

### off(*\texttt{args})

If no arguments are specified, turn all the LEDs off. If arguments are specified, they must be the indexes of the LEDs you wish to turn off. For example:

```python
from gpiozero import LEDBoard

leds = LEDBoard(2, 3, 4, 5)
leds.on()  # turn on all LEDs
leds.off(0)  # turn off the first LED (pin 2)
leds.off(-1)  # turn off the last LED (pin 5)
leds.off(1, 2)  # turn off the middle LEDs (pins 3 and 4)
leds.on()  # turn on all LEDs
```

If `blink()` (page 142) is currently active, it will be stopped first.

**Parameters**  
\texttt{args (int)} – The index(es) of the LED(s) to turn off. If no indexes are specified turn off all LEDs.

### on(*\texttt{args})

If no arguments are specified, turn all the LEDs on. If arguments are specified, they must be the indexes of the LEDs you wish to turn on. For example:

```python
from gpiozero import LEDBoard

leds = LEDBoard(2, 3, 4, 5)
leds.on(0)  # turn on the first LED (pin 2)
leds.on(-1)  # turn on the last LED (pin 5)
leds.on(1, 2)  # turn on the middle LEDs (pins 3 and 4)
leds.off()  # turn off all LEDs
leds.on()  # turn on all LEDs
```

If `blink()` (page 142) is currently active, it will be stopped first.

**Parameters**  
\texttt{args (int)} – The index(es) of the LED(s) to turn on. If no indexes are specified turn on all LEDs.

### pulse(fade_in_time=int, fade_out_time=int, n=None, background=True)

Make all LEDs fade in and out repeatedly. Note that this method will only work if the `pwm` parameter was `True` at construction time.

**Parameters**

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

If no arguments are specified, toggle the state of all LEDs. If arguments are specified, they must be the indexes of the LEDs you wish to toggle. For example:

```python
from gpiozero import LEDBoard

leds = LEDBoard(2, 3, 4, 5)
leds.toggle(0)  # turn on the first LED (pin 2)
leds.toggle(-1) # turn on the last LED (pin 5)
leds.toggle()   # turn the first and last LED off, and the middle pair on
```

If `blink()` (page 142) is currently active, it will be stopped first.

Parameters `args` (`int`\(^{601}\)) – The index(es) of the LED(s) to toggle. If no indexes are specified toggle the state of all LEDs.

### 16.1.2 LEDBarGraph

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

Extends `LEDCollection` (page 164) 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:

```python
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 all other output devices, `source` (page 145) and `values` (page 145) are supported:

```python
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
pause()
```

Parameters

- `*pins` – Specify the GPIO pins that the LEDs of the bar graph are attached to. See `Pin Numbering` (page 3) for valid pin numbers. You can designate as many pins as necessary.
- `pwm` (`bool`\(^{602}\)) – If `True`\(^{603}\), construct `PWMLED` (page 113) instances for each pin. If `False`\(^{604}\) (the default), construct regular `LED` (page 111) instances. This parameter can only be specified as a keyword parameter.

---

\(^{601}\)https://docs.python.org/3.5/library/functions.html#int

\(^{602}\)https://docs.python.org/3.5/library/functions.html#bool

\(^{603}\)https://docs.python.org/3.5/library/constants.html#True

\(^{604}\)https://docs.python.org/3.5/library/constants.html#False
• **active_high** (*bool* 605) – If *True* 606 (the default), the `on()` method will set all the associated pins to HIGH. If *False* 607, the `on()` method will set all pins to LOW (the `off()` method always does the opposite). This parameter can only be specified as a keyword parameter.

• **initial_value** (*float* 608) – The initial `value` (page 145) of the graph given as a float between -1 and +1. Defaults to 0.0. This parameter can only be specified as a keyword parameter.

• **pin_factory** (*Factory* (page 198) or *None* 609) – See API - Pins (page 195) for more information (this is an advanced feature which most users can ignore).

**lit_count**

The number of LEDs on the bar graph actually lit up. Note that just like `value` (page 145), 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 145).

**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:

```python
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 145) is effectively -1 < `value` <= 1.

**values**

An infinite iterator of values read from `value` (page 145).

### 16.1.3 ButtonBoard

```python
class gpiozero.ButtonBoard(*pins, pull_up=True, active_state=None, bounce_time=None, hold_time=1, hold_repeat=False, pin_factory=None, **named_pins)
```  

Extends `CompositeDevice` (page 164) and represents a generic button board or collection of buttons. The `value` (page 147) of the button board is a tuple of all the buttons states. This can be used to control all the LEDs in a `LEDBoard` (page 141) with a `ButtonBoard` (page 145):

```python
from gpiozero import LEDBoard, ButtonBoard
from signal import pause

leds = LEDBoard(2, 3, 4, 5)
btns = ButtonBoard(6, 7, 8, 9)
leds.source = btns.values
pause()
```
Alternatively you could represent the number of pressed buttons with an LEDBarGraph (page 144):

```python
from gpiozero import LEDBarGraph, ButtonBoard
from signal import pause

graph = LEDBarGraph(2, 3, 4, 5)
btns = ButtonBoard(6, 7, 8, 9)
graph.source = (sum(value) for value in btn.values)
pause()
```

Parameters

- **pins** – Specify the GPIO pins that the buttons of the board are attached to. See Pin Numbering (page 3) for valid pin numbers. You can designate as many pins as necessary.
- **pull_up** (`bool` or `None`) – 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. If None, the pin will be floating, so it must be externally pulled up or down and the active_state parameter must be set accordingly.
- **active_state** (`bool` or `None`) – See description under InputDevice (page 107) for more information.
- **bounce_time** (`float`) – If None (the default), no software bounce compensation will be performed. Otherwise, this is the length of time (in seconds) that the buttons will ignore changes in state after an initial change.
- **hold_time** (`float`) – The length of time (in seconds) to wait after any button is pushed, until executing the when_held handler. Defaults to 1.
- **hold_repeat** (`bool`) – If True, the when_held handler will be repeatedly executed as long as any buttons remain held, every hold_time seconds. If False (the default) the when_held handler will be only be executed once per hold.
- **pin_factory** (Factory (page 198) or `None`) – See API - Pins (page 195) 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_press(timeout=None)
```

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

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

is_pressed
    Composite devices are considered “active” if any of their constituent devices have a “truthy” value.

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

value
    A namedtuple containing a value for each subordinate device. Devices with names will be represented as named elements. Unnamed devices will have a unique name generated for them, and they will appear in the position they appeared in the constructor.

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.

16.1.4 TrafficLights

class gpiozero.TrafficLights (red, amber, green, *, yellow=None, pwm=False, initial_value=False, pin_factory=None)
    Extends LEDBoard (page 141) 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:

    ```python
    from gpiozero import TrafficLights
    traffic = TrafficLights(2, 3, 4)
    traffic.amber.on()
    ```

    Parameters

    * red (int or str) – The GPIO pin that the red LED is attached to. See Pin Numbering (page 3) for valid pin numbers.
• **amber** *(int, str or None)* – The GPIO pin that the amber LED is attached to. See *Pin Numbering* (page 3) for valid pin numbers.

• **yellow** *(int or str or None)* – 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. See *Pin Numbering* (page 3) for valid pin numbers.

• **green** *(int or str)* – The GPIO pin that the green LED is attached to. See *Pin Numbering* (page 3) for valid pin numbers.

• **pwm** *(bool)* – If **True**, construct **PWMLED** (page 113) instances to represent each LED. If **False** (the default), construct regular **LED** (page 111) instances.

• **initial_value** *(bool or None)* – 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 or None)* – See *API - Pins* (page 195) for more information (this is an advanced feature which most users can ignore).

red
The red **LED** (page 111) or **PWMLED** (page 113).

amber
The amber **LED** (page 111) or **PWMLED** (page 113). Note that this attribute will not be present when the instance is constructed with the **yellow** keyword parameter.

yellow
The yellow **LED** (page 111) or **PWMLED** (page 113). Note that this attribute will only be present when the instance is constructed with the **yellow** keyword parameter.

green
The green **LED** (page 111) or **PWMLED** (page 113).

### 16.1.5 TrafficLightsBuzzer

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

Extends **CompositeOutputDevice** (page 164) and is a generic class for HATs with traffic lights, a button and a buzzer.

**Parameters**

• **lights** *(TrafficLights (page 147))* – An instance of **TrafficLights** (page 147) representing the traffic lights of the HAT.

• **buzzer** *(Buzzer (page 117))* – An instance of **Buzzer** (page 117) representing the buzzer on the HAT.
• **button** ([Button](page 93)) – An instance of [Button](page 93) representing the button on the HAT.

• **pin_factory** ([Factory](page 198) or [None](#None)) – See API - Pins (page 195) for more information (this is an advanced feature which most users can ignore).

**lights**
The [TrafficLights](page 147) instance passed as the `lights` parameter.

**buzzer**
The [Buzzer](page 117) instance passed as the `buzzer` parameter.

**button**
The [Button](page 93) instance passed as the `button` parameter.

### 16.1.6 PiHutXmasTree

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

Extends [LEDBoard](page 141) for The Pi Hut’s Xmas board: a 3D Christmas tree board with 24 red LEDs and a white LED as a star on top.

The 24 red LEDs can be accessed through the attributes led0, led1, led2, and so on. The white star LED is accessed through the `star` (page 150) attribute. Alternatively, as with all descendents of [LEDBoard](page 141), you can treat the instance as a sequence of LEDs (the first element is the `star` (page 150)).

The Xmas Tree board pins are fixed and therefore there’s no need to specify them when constructing this class. The following example turns all the LEDs on one at a time:

```python
from gpiozero import PiHutXmasTree
from time import sleep

tree = PiHutXmasTree()

for light in tree:
    light.on()
    sleep(1)
```

The following example turns the star LED on and sets all the red LEDs to flicker randomly:

```python
from gpiozero import PiHutXmasTree
from gpiozero.tools import random_values
from signal import pause

tree = PiHutXmasTree(pwm=True)
tree.star.on()

for led in tree[1:]:
    led.source_delay = 0.1
    led.source = random_values()

pause()
```

**Parameters**

• **pwm** ([bool](#bool)) – If [True](#True), construct [PWMLED](page 113) instances for each pin. If [False](#False) (the default), construct regular [LED](page 111) instances.

---

653 https://docs.python.org/3.5/library/constants.html#None
654 https://thepihut.com/xmas
655 https://docs.python.org/3.5/library/functions.html#bool
656 https://docs.python.org/3.5/library/functions.html#True
657 https://docs.python.org/3.5/library/constants.html#False
• **initial_value** ([`bool`][658] or [None][659]) – If [False][660] (the default), all LEDs will be off initially. If [None][661], each device will be left in whatever state the pin is found in when configured for output (warning: this can be on). If [True][662], the device will be switched on initially.

• **pin_factory** ([`Factory`][737] (page 198) or [None][663]) – See [API - Pins](page 195) for more information (this is an advanced feature which most users can ignore).

**star**

Returns the [LED](page 111) or [PWMLED](page 113) representing the white star on top of the tree.

**led0, led1, led2, ...**

Returns the [LED](page 111) or [PWMLED](page 113) representing one of the red LEDs. There are actually 24 of these properties named led0, led1, and so on but for the sake of brevity we represent all 24 under this section.

### 16.1.7 LedBorg

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

Extends [RGBLED](page 115) for the PiBorg LedBorg[664]: 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** ([Color][665] or [tuple][666]) – The initial color for the LedBorg. Defaults to black (0, 0, 0).

• **pwm** ([`bool`][667]) – If [True][668] (the default), construct [PWMLED](page 113) for each component of the LedBorg. If [False][669], construct regular [LED](page 111) instances, which prevents smooth color graduations.

• **pin_factory** ([`Factory`][737] or [None][663]) – See [API - Pins](page 195) for more information (this is an advanced feature which most users can ignore).

### 16.1.8 PiLiter

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

Extends [LEDBoard](page 141) for the Ciseco Pi-LITEr[671]: a strip of 8 very bright LEDs.

[658]: https://docs.python.org/3.5/library/functions.html#bool
[659]: https://docs.python.org/3.5/library/constants.html#None
[660]: https://docs.python.org/3.5/library/constants.html#False
[661]: https://docs.python.org/3.5/library/constants.html#None
[662]: https://docs.python.org/3.5/library/constants.html#True
[663]: https://docs.python.org/3.5/library/constants.html#None
[664]: https://www.piborg.org/ledborg
[666]: https://docs.python.org/3.5/library/stdtypes.html#tuple
[667]: https://docs.python.org/3.5/library/functions.html#bool
[668]: https://docs.python.org/3.5/library/constants.html#True
[669]: https://docs.python.org/3.5/library/constants.html#False
[670]: https://docs.python.org/3.5/library/constants.html#None
[671]: http://shop.ciseco.co.uk/pi-liter-8-led-strip-for-the-raspberry-pi/
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:

```python
from gpiozero import PiLiter
lite = PiLiter()
lite.on()
```

**Parameters**

- **pwm (bool)** – If True, construct PWMLED (page 113) instances for each pin. If False (the default), construct regular LED (page 111) instances.
- **initial_value (bool or None)** – If False (the default), all LEDs will be off initially. If None, each LED will be left in whatever state the pin is found in when configured for output (warning: this can be on). If True, each LED will be switched on initially.
- **pin_factory (Factory or None)** – See API - Pins (page 195) for more information (this is an advanced feature which most users can ignore).

### 16.1.9 PiLiterBarGraph

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

Extends LEDBarGraph (page 144) to treat the Ciseco Pi-LITEr as an 8-segment bar graph. The Pi-LITEr pins are fixed and therefore there’s no need to specify them when constructing this class. The following example sets the graph value to 0.5:

```python
from gpiozero import PiLiterBarGraph
graph = PiLiterBarGraph()
graph.value = 0.5
```

**Parameters**

- **pwm (bool)** – If True, construct PWMLED (page 113) instances for each pin. If False (the default), construct regular LED (page 111) instances.
- **initial_value (float)** – The initial value of the graph given as a float between -1 and +1. Defaults to 0.0.
- **pin_factory (Factory or None)** – See API - Pins (page 195) for more information (this is an advanced feature which most users can ignore).
16.1.10 PiTraffic

```python
from gpiozero import PiTraffic
traffic = PiTraffic()
traffic.amber.on()
```

Extends TrafficLights (page 147) for the Low Voltage Labs PI-TRAFFIC vertical traffic lights board when attached to GPIO pins 9, 10, and 11.

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

Parameters

- `pwm (bool)` – If `True`, construct PWMLED (page 113) instances to represent each LED. If `False` (the default), construct regular LED (page 111) instances.
- `initial_value (bool)` – If `False` (the default), all LEDs will be off initially. If `None`, each device will be left in whatever state the pin is found in when configured for output (warning: this can be on). If `True`, the device will be switched on initially.
- `pin_factory (Factory or None)` – See API - Pins (page 195) for more information (this is an advanced feature which most users can ignore).

16.1.11 PiStop

```python
from gpiozero import PiStop
traffic = PiStop('A+')
traffic.amber.on()
```

Extends TrafficLights (page 147) for the PiHardware Pi-Stop: a vertical traffic lights board.

The following example turns on the amber LED on a Pi-Stop connected to location A+:

Parameters

- `location (str)` – The location on the GPIO header to which the Pi-Stop is connected. Must be one of: A, A+, B, B+, C, D.
- `pwm (bool)` – If `True`, construct PWMLED (page 113) instances to represent each LED. If `False` (the default), construct regular LED (page 111) instances.
• **initial_value** *(bool)* – If False (the default), all LEDs will be off initially. If None, each device will be left in whatever state the pin is found in when configured for output (warning: this can be on). If True, the device will be switched on initially.

• **pin_factory** *(Factory (page 198) or None)* – See API - Pins (page 195) for more information (this is an advanced feature which most users can ignore).

### 16.1.12 FishDish

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

Extends `TrafficLightsBuzzer` (page 148) for the Pi Supply FishDish: traffic light LEDs, a button and a buzzer.

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

```python
from gpiozero import FishDish

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

**Parameters**

• **pwm** *(bool)* – If True, construct `PWMLED` (page 113) instances to represent each LED. If False (the default), construct regular `LED` (page 111) instances.

• **pin_factory** *(Factory (page 198) or None)* – See API - Pins (page 195) for more information (this is an advanced feature which most users can ignore).

### 16.1.13 TrafficHat

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

Extends `TrafficLightsBuzzer` (page 148) for the Ryanteck Traffic HAT: traffic light LEDs, a button and a buzzer.

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

```python
from gpiozero import TrafficHat

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

**Parameters**

702 https://docs.python.org/3.5/library/functions.html#bool
703 https://docs.python.org/3.5/library/constants.html#False
704 https://docs.python.org/3.5/library/constants.html#None
705 https://docs.python.org/3.5/library/constants.html#True
706 https://docs.python.org/3.5/library/constants.html#None
707 https://www.pi-supply.com/product/fish-dish-raspberry-pi-led-buzzer-board/
708 https://docs.python.org/3.5/library/functions.html#bool
709 https://docs.python.org/3.5/library/constants.html#True
710 https://docs.python.org/3.5/library/constants.html#False
711 https://docs.python.org/3.5/library/constants.html#None
712 https://ryanteck.uk/hats/1-traffichat-0635648607122.html
• **pwm** *(bool)* – If **True**, construct `PWMLED` (page 113) instances to represent each LED. If **False** (the default), construct regular `LED` (page 111) instances.

• **pin_factory** *(Factory or None)* – See `API - Pins` (page 195) for more information (this is an advanced feature which most users can ignore).

### 16.1.14 JamHat

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

Extends `CompositeOutputDevice` (page 164) for the ModMyPi JamHat board.

There are 6 LEDs, two buttons and a tonal buzzer. The pins are fixed. Usage:

```python
from gpiozero import JamHat

hat = JamHat()

hat.button_1.wait_for_press()
hat.lights_1.on()
hat.buzzer.play('C4')
hat.button_2.wait_for_press()
hat.off()
```

**Parameters**

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

• **pin_factory** *(Factory or None)* – See `API - Pins` (page 195) for more information (this is an advanced feature which most users can ignore).

**lights_1, lights_2**

Two `LEDBoard` (page 141) instances representing the top (lights_1) and bottom (lights_2) rows of LEDs on the JamHat.

**red, yellow, green**

`LED` (page 111) or `PWMLED` (page 113) instances representing the red, yellow, and green LEDs along the top row.

**button_1, button_2**

The left (button_1) and right (button_2) `Button` (page 93) objects on the JamHat.

**buzzer**

The `Buzzer` (page 117) at the bottom right of the JamHat.

**off**

Turns all the LEDs off and stops the buzzer.

**on**

Turns all the LEDs on and makes the buzzer play its mid tone.

---

713 https://docs.python.org/3.5/library/functions.html#bool
714 https://docs.python.org/3.5/library/constants.html#True
715 https://docs.python.org/3.5/library/constants.html#False
716 https://docs.python.org/3.5/library/constants.html#None
717 https://www.modmypi.com/jam-hat
718 https://docs.python.org/3.5/library/functions.html#bool
719 https://docs.python.org/3.5/library/constants.html#True
720 https://docs.python.org/3.5/library/constants.html#False
721 https://docs.python.org/3.5/library/constants.html#None
16.1.15 Robot

class gpiozero.Robot (left, right, *, pwm=True, pin_factory=None)
    Extends CompositeDevice (page 164) to represent a generic dual-motor robot.

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

    from gpiozero import Robot
    robot = Robot(left=(4, 14), right=(17, 18))
    robot.forward()

Parameters

- **left** *(tuple)* – A tuple of two (or three) GPIO pins representing the forward and backward inputs of the left motor’s controller. Use three pins if your motor controller requires an enable pin.

- **right** *(tuple)* – A tuple of two (or three) GPIO pins representing the forward and backward inputs of the right motor’s controller. Use three pins if your motor controller requires an enable pin.

- **pwm** *(bool)* – If True (the default), construct PWMOutputDevice (page 128) instances for the motor controller pins, allowing both direction and variable speed control. If False, construct DigitalOutputDevice (page 127) instances, allowing only direction control.

- **pin_factory** *(Factory or None)* – See API - Pins (page 195) for more information (this is an advanced feature which most users can ignore).

**left_motor**

The Motor (page 120) on the left of the robot.

**right_motor**

The Motor (page 120) on the right of the robot.

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

Drive the robot backward by running both motors backward.

Parameters

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

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

reverse()

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

right(speed=1)

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

Parameters

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

stop()

Stop the robot.

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.

16.1.16 PhaseEnableRobot

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

Extends CompositeDevice (page 164) 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:

```python
from gpiozero import PhaseEnableRobot

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

731 https://docs.python.org/3.5/library/functions.html#float
732 https://docs.python.org/3.5/library/functions.html#float
733 https://docs.python.org/3.5/library/functions.html#float
734 https://docs.python.org/3.5/library/functions.html#float
735 https://docs.python.org/3.5/library/functions.html#float
Parameters

- **left** *(tuple)* – A tuple of two GPIO pins representing the phase and enable inputs of the left motor’s controller.

- **right** *(tuple)* – A tuple of two GPIO pins representing the phase and enable inputs of the right motor’s controller.

- **pwm** *(bool)* – If True (the default), construct PWMOutputDevice instances for the motor controller’s enable pins, allowing both direction and variable speed control. If False, construct DigitalOutputDevice instances, allowing only direction control.

- **pin_factory** *(Factory or None)* – See API - Pins for more information (this is an advanced feature which most users can ignore).

### left_motor

The PhaseEnableMotor (page 122) on the left of the robot.

### right_motor

The PhaseEnableMotor (page 122) on the right of the robot.

### backward *(speed=1)*

Drive the robot backward by running both motors backward.

**Parameters**

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

### forward *(speed=1)*

Drive the robot forward by running both motors forward.

**Parameters**

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

### left *(speed=1)*

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

**Parameters**

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

### reverse ()

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

### right *(speed=1)*

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

**Parameters**

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

### stop ()

Stop the robot.

### 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.

---

[736](https://docs.python.org/3.5/library/stdtypes.html#tuple)
[737](https://docs.python.org/3.5/library/stdtypes.html#tuple)
[738](https://docs.python.org/3.5/library/functions.html#bool)
[739](https://docs.python.org/3.5/library/constants.html#True)
[740](https://docs.python.org/3.5/library/constants.html#False)
[741](https://docs.python.org/3.5/library/constants.html#None)
[742](https://docs.python.org/3.5/library/functions.html#float)
[743](https://docs.python.org/3.5/library/functions.html#float)
[744](https://docs.python.org/3.5/library/functions.html#float)
[745](https://docs.python.org/3.5/library/functions.html#float)
16.1.17 RyanteckRobot
class gpiozero.RyanteckRobot(*, pwm=True, pin_factory=None)
    Extends Robot (page 155) for the Ryanteck motor controller board.

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

```python
from gpiozero import RyanteckRobot
robot = RyanteckRobot()
robot.forward()
```

Parameters

- **pwm** (`bool`) – If `True` (the default), construct `PWMOutputDevice` (page 128) instances for the motor controller pins, allowing both direction and variable speed control. If `False`, construct `DigitalOutputDevice` (page 127) instances, allowing only direction control.

- **pin_factory** (`Factory` or `None`) – See API - Pins (page 195) for more information (this is an advanced feature which most users can ignore).

16.1.18 CamJamKitRobot
class gpiozero.CamJamKitRobot(*, pwm=True, pin_factory=None)
    Extends Robot (page 155) for the CamJam #3 EduKit 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:

```python
from gpiozero import CamJamKitRobot
robot = CamJamKitRobot()
robot.forward()
```

Parameters

- **pwm** (`bool`) – If `True` (the default), construct `PWMOutputDevice` (page 128) instances for the motor controller pins, allowing both direction and variable speed control. If `False`, construct `DigitalOutputDevice` (page 127) instances, allowing only direction control.

- **pin_factory** (`Factory` or `None`) – See API - Pins (page 195) for more information (this is an advanced feature which most users can ignore).

16.1.19 PololuDRV8835Robot
class gpiozero.PololuDRV8835Robot(*, pwm=True, pin_factory=None)
    Extends PhaseEnableRobot (page 156) for the Pololu DRV8835 Dual Motor Driver Kit.

---

746 https://ryanteck.uk/add-ons/6-ryanteck-rpi-motor-controller-board-0635648607160.html
747 https://docs.python.org/3.5/library/functions.html#bool
748 https://docs.python.org/3.5/library/constants.html#True
749 https://docs.python.org/3.5/library/constants.html#False
750 https://docs.python.org/3.5/library/constants.html#None
751 http://camjam.me/?page_id=1035
752 https://docs.python.org/3.5/library/functions.html#bool
753 https://docs.python.org/3.5/library/constants.html#True
754 https://docs.python.org/3.5/library/constants.html#False
755 https://docs.python.org/3.5/library/constants.html#None
756 https://www.pololu.com/product/2753
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:

```python
from gpiozero import PololuDRV8835Robot
robot = PololuDRV8835Robot()
robot.forward()
```

**Parameters**

- **pwm** *(bool)* – If True (the default), construct `PWMOutputDevice` (page 128) instances for the motor controller’s enable pins, allowing both direction and variable speed control. If False, construct `DigitalOutputDevice` (page 127) instances, allowing only direction control.
- **pin_factory** *(Factory (page 198) or None)* – See API - Pins (page 195) for more information (this is an advanced feature which most users can ignore).

### 16.1.20 Energenie

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

Extends `Device` (page 175) to represent an Energenie socket controller.

This class is constructed with a socket number and an optional initial state (defaults to False, meaning off). Instances of this class can be used to switch peripherals on and off. For example:

```python
from gpiozero import Energenie
lamp = Energenie(1)
lamp.on()
```

**Parameters**

- **socket** *(int)* – Which socket this instance should control. This is an integer number between 1 and 4.
- **initial_value** *(bool)* – The initial state of the socket. As Energenie sockets provide no means of reading their state, you must provide an initial state for the socket, which will be set upon construction. This defaults to False which will switch the socket off.
- **pin_factory** *(Factory (page 198) or None)* – See API - Pins (page 195) for more information (this is an advanced feature which most users can ignore).

```python
lamp.off()  # Turns the socket off.
lamp.on()   # Turns the socket on.
lamp.socket # Returns the socket number.
```

---

757 https://docs.python.org/3.5/library/functions.html#bool
758 https://docs.python.org/3.5/library/constants.html#True
759 https://docs.python.org/3.5/library/constants.html#False
760 https://docs.python.org/3.5/library/constants.html#None
761 https://energenie4u.co.uk/index.php/catalogue/product/ENER002-2PI
762 https://docs.python.org/3.5/library/constants.html#False
763 https://docs.python.org/3.5/library/functions.html#import
764 https://docs.python.org/3.5/library/functions.html#bool
765 https://docs.python.org/3.5/library/constants.html#False
766 https://docs.python.org/3.5/library/constants.html#None

value

Returns `True` if the socket is on and `False` if the socket is off. Setting this property changes the state of the socket.

16.1.21 StatusZero

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

Extends LEDBoard (page 141) for The Pi Hut’s STATUS Zero: 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:

```python
from gpiozero import StatusZero
status = StatusZero('wifi', 'raining')
status.wifi.green.on()
status.raining.red.on()
```

Each designated label will contain two LED (page 111) objects named “red” and “green”.

Parameters

- *labels (str) – 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 or None) – See API - Pins (page 195) for more information (this is an advanced feature which most users can ignore).

your-label-here, your-label-here, ...

This entry represents one of the three labelled attributes supported on the STATUS Zero board. It is an LEDBoard (page 141) which contains:

red

The LED (page 111) or PWMLED (page 113) representing the red LED next to the label.

green

The LED (page 111) or PWMLED (page 113) representing the green LED next to the label.

16.1.22 StatusBoard

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

Extends CompositeOutputDevice (page 164) for The Pi Hut’s STATUS 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:

```python
from gpiozero import StatusBoard
status = StatusBoard('wifi', 'raining')
```
Each designated label will contain a “lights” `LEDBoard` (page 141) containing two `LED` (page 111) objects named “red” and “green”, and a `Button` (page 93) object named “button”.

**Parameters**

- **labels** *(str)* – 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 198) or None)* – See API - Pins (page 195) for more information (this is an advanced feature which most users can ignore).

```python
status.wifi.lights.green.on()
status.wifi.button.when_pressed = status.wifi.lights.toggle
```

This entry represents one of the five labelled attributes supported on the STATUS board. It is an `CompositeOutputDevice` (page 164) which contains:

- **lights**
  - A `LEDBoard` (page 141) representing the lights next to the label. It contains:
    - **red**
      - The `LED` (page 111) or `PWMLED` (page 113) representing the red LED next to the label.
    - **green**
      - The `LED` (page 111) or `PWMLED` (page 113) representing the green LED next to the label.
  - **button**
    - A `Button` (page 93) representing the button next to the label.

### 16.1.23 SnowPi

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

Extends `LEDBoard` (page 141) for the Ryanteck SnowPi board.

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

```python
from gpiozero import SnowPi

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

**Parameters**

- **pwm** *(bool)* – If `True`, construct `PWMLED` (page 113) instances to represent each LED. If `False` (the default), construct regular `LED` (page 111) instances.

- **initial_value** *(bool)* – If `False` (the default), all LEDs will be off initially. If `None`, each device will be left in whatever state the pin is found in when configured.
for output (warning: this can be on). If True\textsuperscript{782}, the device will be switched on initially.

- **pin\_factory** (Factory \textsuperscript{783} or None\textsuperscript{784}) – See API - Pins (page \textsuperscript{785}) for more information (this is an advanced feature which most users can ignore).

**arms**

A LEDBoard (page 141) representing the arms of the snow man. It contains the following attributes:

- **left, right**
  
  Two LEDBoard (page 141) objects representing the left and right arms of the snow-man. They contain:

  - **top, middle, bottom**
    
    The LED (page 111) or PWMLED (page 113) down the snow-man’s arms.

**eyes**

A LEDBoard (page 141) representing the eyes of the snow-man. It contains:

- **left, right**
  
  The LED (page 111) or PWMLED (page 113) for the snow-man’s eyes.

**nose**

The LED (page 111) or PWMLED (page 113) for the snow-man’s nose.

### 16.1.24 PumpkinPi

class gpiozero.PumpkinPi(*, pwm=False, initial_value=False, pin\_factory=None)

Extends LEDBoard (page 141) for the ModMyPi PumpkinPi\textsuperscript{784} board.

There are twelve LEDs connected up to individual pins, so for the PumpkinPi the pins are fixed. For example:

```python
from gpiozero import PumpkinPi

pumpkin = PumpkinPi(pwm=True)
pumpkin.sides.pulse()
pumpkin.off()
```

Parameters

- **pwm** (bool\textsuperscript{785}) – If True\textsuperscript{786}, construct PWMLED (page 113) instances to represent each LED. If False\textsuperscript{787} (the default), construct regular LED (page 111) instances

- **initial\_value** (bool\textsuperscript{788} or None\textsuperscript{789}) – If False\textsuperscript{790} (the default), all LEDs will be off initially. If None\textsuperscript{791}, each device will be left in whatever state the pin is found in when configured for output (warning: this can be on). If True\textsuperscript{792}, the device will be switched on initially.

- **pin\_factory** (Factory \textsuperscript{783} or None\textsuperscript{793}) – See API - Pins (page \textsuperscript{794}) for more information (this is an advanced feature which most users can ignore).

\textsuperscript{782} https://docs.python.org/3.5/library/constants.html#True
\textsuperscript{783} https://docs.python.org/3.5/library/constants.html#Factory
\textsuperscript{784} https://www.modmypi.com/halloween-pumpkin-programmable-kit
\textsuperscript{785} https://docs.python.org/3.5/library/functions.html#bool
\textsuperscript{786} https://docs.python.org/3.5/library/constants.html#False
\textsuperscript{787} https://docs.python.org/3.5/library/functions.html#true
\textsuperscript{788} https://docs.python.org/3.5/library/functions.html#false
\textsuperscript{789} https://docs.python.org/3.5/library/constants.html#None
\textsuperscript{790} https://docs.python.org/3.5/library/constants.html#false
\textsuperscript{791} https://docs.python.org/3.5/library/constants.html#true
\textsuperscript{792} https://docs.python.org/3.5/library(constants.html#true
\textsuperscript{793} https://docs.python.org/3.5/library/constants.html#false
\textsuperscript{794} https://docs.python.org/3.5/library/constants.html#None
sides
A LEDBoard (page 141) representing the LEDs around the edge of the pumpkin. It contains:

left, right
Two LEDBoard (page 141) instances representing the LEDs on the left and right sides of the pumpkin. They each contain:

top, midtop, middle, midbottom, bottom
Each LED (page 111) or PWMLED (page 113) around the specified side of the pumpkin.

eyes
A LEDBoard (page 141) representing the eyes of the pumpkin. It contains:

left, right
The LED (page 111) or PWMLED (page 113) for each of the pumpkin’s eyes.

16.2 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.

### 16.2.1 LEDCollection

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

Extends `CompositeOutputDevice` (page 164). Abstract base class for `LEDBoard` (page 141) and `LEDBarGraph` (page 144).

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

### 16.2.2 CompositeOutputDevice

**class** `gpiozero.CompositeOutputDevice(*args, _order=None, pin_factory=None, **kwargs)`

Extends `CompositeDevice` (page 164) with `on()` (page 164), `off()` (page 164), and `toggle()` (page 164) methods for controlling subordinate output devices. Also extends `value` (page 164) to be writeable.

- **Parameters**
  - `*args` ([Device](page 175)) – The un-named devices that belong to the composite device. The `value` (page 175) attributes of these devices will be represented within the composite device’s tuple `value` (page 164) in the order specified here.
  - `_order` ([list](https://docs.python.org/3.5/library/stdtypes.html#list) or **None**) – If specified, this is the order of named items specified by keyword arguments (to ensure that the `value` (page 164) 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 198) or **None**) – See API - Pins (page 195) for more information (this is an advanced feature which most users can ignore).
  - `**kwargs` ([Device](page 175)) – The named devices that belong to the composite device. These devices will be accessible as named attributes on the resulting device, and their `value` (page 164) attributes will be accessible as named elements of the composite device’s tuple `value` (page 164).

- `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.

### 16.2.3 CompositeDevice

**class** `gpiozero.CompositeDevice(*args, _order=None, pin_factory=None, **kwargs)`

Extends `Device` (page 175). Represents a device composed of multiple devices like simple HATs, H-bridge motor controllers, robots composed of multiple motors, etc.

---

794 https://docs.python.org/3.5/library/stdtypes.html#list
795 https://docs.python.org/3.5/library/constants.html#None
796 https://docs.python.org/3.5/library/constants.html#None
The constructor accepts subordinate devices as positional or keyword arguments. Positional arguments form unnamed devices accessed by treating the composite device as a container, while keyword arguments are added to the device as named (read-only) attributes.

For example:

```python
>>> from gpiozero import *
>>> d = CompositeDevice(LED(2), LED(3), LED(4), btn=Button(17))
>>> d[0]
<gpiozero.LED object on pin GPIO2, active_high=True, is_active=False>
>>> d[1]
<gpiozero.LED object on pin GPIO3, active_high=True, is_active=False>
>>> d[2]
<gpiozero.LED object on pin GPIO4, active_high=True, is_active=False>
>>> d.btn
<gpiozero.Button object on pin GPIO17, pull_up=True, is_active=False>
>>> d.value
CompositeDeviceValue(device_0=False, device_1=False, device_2=False, btn=False)
```

**Parameters**

- `*args` (Device (page 175)) – The un-named devices that belong to the composite device. The value (page 166) attributes of these devices will be represented within the composite device’s tuple `value` (page 166) in the order specified here.

- `_order` (list797) or `None798` – If specified, this is the order of named items specified by keyword arguments (to ensure that the value (page 166) 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 198) or `None799`) – See API - Pins (page 195) for more information (this is an advanced feature which most users can ignore).

- `**kwargs` (Device (page 175)) – The named devices that belong to the composite device. These devices will be accessible as named attributes on the resulting device, and their value (page 166) attributes will be accessible as named elements of the composite device’s tuple `value` (page 166).

**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:

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

---

797 https://docs.python.org/3.5/library/stdtypes.html#list
798 https://docs.python.org/3.5/library/constants.html#None
799 https://docs.python.org/3.5/library/constants.html#None
Device (page 175) descendents can also be used as context managers using the `with` statement. For example:

```python
>>> 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 165) method). Once a device is closed you can no longer use any other methods or properties to control or query the device.

is_active

Composite devices are considered “active” if any of their constituent devices have a “truthy” value.

namedtuple

The `namedtuple()` type constructed to represent the value of the composite device. The `value` (page 166) attribute returns values of this type.

value

A `namedtuple()` containing a value for each subordinate device. Devices with names will be represented as named elements. Unnamed devices will have a unique name generated for them, and they will appear in the position they appeared in the constructor.
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!

### 17.1 Regular Classes

The following classes are intended for general use with the devices they are named after. All classes in this section are concrete (not abstract).

#### 17.1.1 TimeOfDay

**class** `gpiozero.TimeOfDay(start_time, end_time, *, utc=True, pin_factory=None)`

Extends `InternalDevice` (page 172) to provide a device which is active when the computer’s clock indicates that the current time is between `start_time` and `end_time` (inclusive) which are `time` instances.

The following example turns on a lamp attached to an `Energenie` (page 159) plug between 7 and 8 AM:

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

lamp = Energenie(1)
morning = TimeOfDay(time(7), time(8))
lamp.source = morning
pause()
```

Note that `start_time` may be greater than `end_time`, indicating a time period which crosses midnight.

**Parameters**

---

804 [https://docs.python.org/3.5/library/datetime.html#datetime.time](https://docs.python.org/3.5/library/datetime.html#datetime.time)
• **start_time** *(time)* – The time from which the device will be considered active.

• **end_time** *(time)* – The time after which the device will be considered inactive.

• **utc** *(bool)* – If True (the default), a naive UTC time will be used for the comparison rather than a local time-zone reading.

• **pin_factory** *(Factory (page 198) or None)* – See API - Pins (page 195) for more information (this is an advanced feature which most users can ignore).

---

**end_time**

The time of day after which the device will be considered inactive.

**start_time**

The time of day after which the device will be considered active.

**utc**

If True, use a naive UTC time reading for comparison instead of a local timezone reading.

**value**

Returns True when the system clock reads between `start_time` (page 168) and `end_time` (page 168), and False otherwise. If `start_time` (page 168) is greater than `end_time` (page 168) (indicating a period that crosses midnight), then this returns True when the current time is greater than `start_time` (page 168) or less than `end_time` (page 168).

---

### 17.1.2 PingServer
def class gpiozero.PingServer(*, host, pin_factory=None)

Extends InternalDevice (page 172) 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 176) to ensure the server is not flooded with pings):

```python
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

pause()
```

**Parameters**

- **host** *(str)* – The hostname or IP address to attempt to ping.

- **pin_factory** *(Factory (page 198) or None)* – See API - Pins (page 195) for more information (this is an advanced feature which most users can ignore).
The hostname or IP address to test whenever `value` (page 169) is queried.

Returns `True` if the host returned a single ping, and `False` otherwise.

### 17.1.3 CPUPTemperature

```python
class gpiozero.CPUTemperature(sensor_file='/sys/class/thermal/thermal_zone0/temp',
                               *, min_temp=0.0, max_temp=100.0, threshold=80.0,
                               pin_factory=None)
```

Extends `InternalDevice` (page 172) 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:

```python
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: {:.1f}°C'.format(cpu.temperature))

graph = LEDBarGraph(5, 6, 13, 19, 25, pwm=True)
graph.source = cpu

pause()
```

**Parameters**

- `sensor_file` (`str`) – The file from which to read the temperature. This defaults to the sysfs file `/sys/class/thermal/thermal_zone0/temp`. Whatever file is specified is expected to contain a single line containing the temperature in millidegrees celsius.
- `min_temp` (`float`) – The temperature at which `value` (page 169) will read 0.0. This defaults to 0.0.
- `max_temp` (`float`) – The temperature at which `value` (page 169) will read 1.0. This defaults to 100.0.
- `threshold` (`float`) – The temperature above which the device will be considered “active”. (see `is_active` (page 169)). This defaults to 80.0.
- `pin_factory` (`Factory` or `None`) – See API - Pins (page 195) for more information (this is an advanced feature which most users can ignore).

**is_active**

Returns `True` when the CPU temperature (page 169) exceeds the `threshold`.

**temperature**

Returns the current CPU temperature in degrees celsius.

---

816 [https://docs.python.org/3.5/library/constants.html#True](https://docs.python.org/3.5/library/constants.html#True)
817 [https://docs.python.org/3.5/library/constants.html#False](https://docs.python.org/3.5/library/constants.html#False)
818 [https://docs.python.org/3.5/library/stdtypes.html#str](https://docs.python.org/3.5/library/stdtypes.html#str)
819 [https://docs.python.org/3.5/library/functions.html#float](https://docs.python.org/3.5/library/functions.html#float)
820 [https://docs.python.org/3.5/library/functions.html#float](https://docs.python.org/3.5/library/functions.html#float)
821 [https://docs.python.org/3.5/library/functions.html#float](https://docs.python.org/3.5/library/functions.html#float)
822 [https://docs.python.org/3.5/library/constants.html#None](https://docs.python.org/3.5/library/constants.html#None)
823 [https://docs.python.org/3.5/library/constants.html#True](https://docs.python.org/3.5/library/constants.html#True)
value

Returns the current CPU temperature as a value between 0.0 (representing the min_temp value) and 1.0 (representing the max_temp value). These default to 0.0 and 100.0 respectively, hence value (page 169) is temperature (page 169) divided by 100 by default.

17.1.4 LoadAverage

class gpiozero.LoadAverage(load_average_file='/proc/loadavg', *, min_load_average=0.0, max_load_average=1.0, threshold=0.8, minutes=5, pin_factory=None)

Extends InternalDevice (page 172) to provide a device which is active when the CPU load average exceeds the threshold value.

The following example plots the load average on an LED bar graph:

```python
from gpiozero import LEDBarGraph, LoadAverage
from signal import pause

la = LoadAverage(min_load_average=0, max_load_average=2)
graph = LEDBarGraph(5, 6, 13, 19, 25, pwm=True)

graph.source = la

pause()
```

Parameters

- **load_average_file** *(str)* – The file from which to read the load average. This defaults to the proc file /proc/loadavg. Whatever file is specified is expected to contain three space-separated load averages at the beginning of the file, representing 1 minute, 5 minute and 15 minute averages respectively.

- **min_load_average** *(float)* – The load average at which value (page 170) will read 0.0. This defaults to 0.0.

- **max_load_average** *(float)* – The load average at which value (page 170) will read 1.0. This defaults to 1.0.

- **threshold** *(float)* – The load average above which the device will be considered “active”. (see is_active (page 170)). This defaults to 0.8.

- **minutes** *(int)* – The number of minutes over which to average the load. Must be 1, 5 or 15. This defaults to 5.

- **pin_factory** *(Factory or None)* – See API - Pins (page 195) for more information (this is an advanced feature which most users can ignore).

is_active

Returns True when the load_average (page 170) exceeds the threshold.

load_average

Returns the current load average.

value

Returns the current load average as a value between 0.0 (representing the min_load_average value) and 1.0 (representing the max_load_average value). These default to 0.0 and 1.0 respectively.

---

824 https://docs.python.org/3.5/library/stdtypes.html#str
825 https://docs.python.org/3.5/library/functions.html#float
826 https://docs.python.org/3.5/library/functions.html#float
827 https://docs.python.org/3.5/library/functions.html#float
828 https://docs.python.org/3.5/library/functions.html#int
829 https://docs.python.org/3.5/library/constants.html#None
830 https://docs.python.org/3.5/library/constants.html#True
17.1.5 DiskUsage

class gpiozero.DiskUsage(filesystem='/', *, threshold=90.0, pin_factory=None)

Extends InternalDevice (page 172) to provide a device which is active when the disk space used exceeds the threshold value.

The following example plots the disk usage on an LED bar graph:

```python
from gpiozero import LEDBarGraph, DiskUsage
from signal import pause

disk = DiskUsage()
print('Current disk usage: {}%'.format(disk.usage))

graph = LEDBarGraph(5, 6, 13, 19, 25, pwm=True)
graph.source = disk

pause()
```

**Parameters**

- **filesystem** (str) – A path within the filesystem for which the disk usage needs to be computed. This defaults to /, which is the root filesystem.
- **threshold** (float) – The disk usage percentage above which the device will be considered “active” (see is_active (page 171)). This defaults to 90.0.
- **pin_factory** (Factory (page 198) or None) – See API - Pins (page 195) for more information (this is an advanced feature which most users can ignore).

**is_active**

Returns True when the disk usage (page 171) exceeds the threshold.

**usage**

Returns the current disk usage in percentage.

**value**

Returns the current disk usage as a value between 0.0 and 1.0 by dividing usage (page 171) by 100.

17.2 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):

---

831 https://docs.python.org/3.5/library/stdtypes.html#str
832 https://docs.python.org/3.5/library/functions.html#float
833 https://docs.python.org/3.5/library/constants.html#None
834 https://docs.python.org/3.5/library/constants.html#True
The following sections document these base classes for advanced users that wish to construct classes for their own devices.

17.2.1 InternalDevice

```python
class gpiozero.InternalDevice(*, pin_factory=None)
```

Extends `Device` (page 175) 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.
The GPIO Zero class hierarchy is quite extensive. It contains several base classes (most of which are documented in their corresponding chapters):

- **Device** (page 175) is the root of the hierarchy, implementing base functionality like `close()` (page 175) and context manager handlers.
- **GPIODevice** (page 108) represents individual devices that attach to a single GPIO pin
- **SPIDevice** (page 139) represents devices that communicate over an SPI interface (implemented as four GPIO pins)
- **InternalDevice** (page 172) represents devices that are entirely internal to the Pi (usually operating system related services)
- **CompositeDevice** (page 164) represents devices composed of multiple other devices like HATs

There are also several mixin classes\(^\text{835}\) 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):

\(^{835}\) https://en.wikipedia.org/wiki/Mixin
18.1 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 175) property, the `value` (page 175) property, and the `close()` (page 175) method).

**pin_factory**

This attribute exists at both a class level (representing the default pin factory used to construct devices when no `pin_factory` parameter is specified), and at an instance level (representing the pin factory that the device was constructed with).

The pin factory provides various facilities to the device including allocating pins, providing low level interfaces (e.g. SPI), and clock facilities (querying and calculating elapsed times).

**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:

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

Device (page 175) descendents can also be used as context managers using the `with` statement. For example:

```python
>>> 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 175) 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 175). Unlike `value` (page 175), this is always a boolean.

**value**

Returns a value representing the device’s state. Frequently, this is a boolean value, or a number

---

https://docs.python.org/3.5/reference/compound_stmts.html#with
https://docs.python.org/3.5/library/constants.html#True
https://docs.python.org/3.5/library/constants.html#False
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.

18.2 ValuesMixin

class gpiozero.ValuesMixin(...)

Adds a \texttt{values} (page 176) property to the class which returns an infinite generator of readings from the \texttt{value} (page 175) property. There is rarely a need to use this mixin directly as all base classes in GPIO Zero include it.

\textbf{Note:} Use this mixin \textit{first} in the parent class list.

\begin{itemize}
\item \texttt{values}
  \begin{itemize}
  \item An infinite iterator of values read from \texttt{value}.
  \end{itemize}
\end{itemize}

18.3 SourceMixin

class gpiozero.SourceMixin(...)

Adds a \texttt{source} (page 176) property to the class which, given an iterable or a ValuesMixin (page 176) descendent, sets \texttt{value} (page 175) 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.

\textbf{Note:} Use this mixin \textit{first} in the parent class list.

\begin{itemize}
\item \texttt{source}
  \begin{itemize}
  \item The iterable to use as a source of values for \texttt{value}.
  \end{itemize}
\item \texttt{source\_delay}
  \begin{itemize}
  \item The delay (measured in seconds) in the loop used to read values from \texttt{source} (page 176). Defaults to 0.01 seconds which is generally sufficient to keep CPU usage to a minimum while providing adequate responsiveness.
  \end{itemize}
\end{itemize}

18.4 SharedMixin

class gpiozero.SharedMixin(...)

This mixin marks a class as \textquotedblleft shared\textquotedblright. In this case, the meta-class (GPIOMeta) will use \texttt{\_shared\_key()} (page 176) 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 \textquotedblleft constructed\textquotedblright in this way.

When \texttt{close()} (page 175) is called, an internal reference counter will be decremented and the instance will only close when it reaches zero.

\begin{itemize}
\item \texttt{classmethod \_shared\_key(*args, **kwargs)}
  \begin{itemize}
  \item Given the constructor arguments, returns an immutable key representing the instance. The default simply assumes all positional arguments are immutable.
  \end{itemize}
\end{itemize}
18.5 EventsMixin

class gpiozero.EventsMixin(...)

Adds edge-detected `when_activated()` (page 177) and `when_deactivated()` (page 177) events to a device based on changes to the `is_active` (page 175) property common to all devices. Also adds `wait_for_active()` (page 177) and `wait_for_inactive()` (page 177) 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* or *None*) – 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* or *None*) – 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.

---

840 https://docs.python.org/3.5/library/functions.html#float
841 https://docs.python.org/3.5/library/constants.html#None
842 https://docs.python.org/3.5/library/constants.html#None
843 https://docs.python.org/3.5/library/functions.html#float
844 https://docs.python.org/3.5/library/constants.html#None
845 https://docs.python.org/3.5/library/constants.html#None
846 https://docs.python.org/3.5/library/constants.html#None
847 https://docs.python.org/3.5/library/constants.html#None
848 https://docs.python.org/3.5/library/constants.html#None
849 https://docs.python.org/3.5/library/constants.html#None
18.6 HoldMixin

class gpiozero.HoldMixin(...)

Extends EventsMixin (page 177) to add the when_held (page 178) event and the machinery to fire that event repeatedly (when hold_repeat (page 178) is True850) at internals defined by hold_time (page 178).

**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 178) event rather than when the device activated, in contrast to active_time (page 177). If the device is not currently held, this is None851.

**hold_repeat**

If True852, when_held (page 178) will be executed repeatedly with hold_time (page 178) 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 178) handler. If hold_repeat (page 178) is True, this is also the length of time between invocations of when_held (page 178).

**is_held**

When True853, the device has been active for at least hold_time (page 178) seconds.

**when_held**

The function to run when the device has remained active for hold_time (page 178) 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 None854 (the default) to disable the event.

---

850 https://docs.python.org/3.5/library/constants.html#True
851 https://docs.python.org/3.5/library/constants.html#None
852 https://docs.python.org/3.5/library/constants.html#True
853 https://docs.python.org/3.5/library/constants.html#True
854 https://docs.python.org/3.5/library/constants.html#None
GPIO Zero includes several utility routines which are intended to be used with the `Source/Values` attributes common to most devices in the library. These utility routines are in the `tools` module of GPIO Zero and are typically imported as follows:

```python
from gpiozero.tools import scaled, negated, all_values
```

Given that `source` and `values` deal with infinite iterators, another excellent source of utilities is the `itertools` module in the standard library.

### 19.1 Single source conversions

**`gpiozero.tools.absoluted(values)`**

Returns `values` with all negative elements negated (so that they’re positive). For example:

```python
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, -1, 1)
led.source = absoluted(motor)
pause()
```

**`gpiozero.tools.booleanized(values, min_value, max_value, hysteresis=0)`**

Returns True for each item in `values` between `min_value` and `max_value`, and False otherwise. `hysteresis` can optionally be used to add hysteresis\(^5\) 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 ¼ and ¾ of its full range:

\(^5\) [https://docs.python.org/3.5/library/itertools.html#module-itertools](https://docs.python.org/3.5/library/itertools.html#module-itertools)

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, 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, 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)
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)

(continues on next page)
gpiozero.tools\.\* \**post\_delayed\*(values, delay)\**
Waits for \*delay\* seconds after returning each item from \*values\*.

gpiozero.tools\.\* \**post\_periodic\_filtered\*(values, repeat\_after, block)\**
After every \*repeat\_after\* items, blocks the next \*block\* items from \*values\*. Note that unlike \*pre\_periodic\_filtered()\* (page 181), \*repeat\_after\* can’t be 0. For example, to block every tenth item read from an ADC:

```python
from gpiozero import MCP3008
from gpiozero.tools import post_periodic_filtered
adc = MCP3008(channel=0)
for value in post_periodic_filtered(adc, 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:

```python
from gpiozero import MCP3008
from gpiozero.tools import pre_periodic_filtered
adc = MCP3008(channel=0)
for value in pre_periodic_filtered(adc, 50, 0):
    print(value)
```

Or to only display every even item read from an ADC:

```python
from gpiozero import MCP3008
from gpiozero.tools import pre_periodic_filtered
adc = MCP3008(channel=0)
for value in pre_periodic_filtered(adc, 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):

```python
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, 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:

```python
from gpiozero import LEDBoard, Button
from gpiozero.tools import queued
from signal import pause

leds = LEDBoard(5, 6, 13, 19, 26)
btn = Button(17)

for i in range(4):
    leds[i].source = queued(leds[i + 1], 5)
    leds[i].source_delay = 0.01

leds[4].source = btn
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:

```python
from gpiozero import MCP3008
from gpiozero.tools import smoothed

adc = MCP3008(channel=0)

for value in smoothed(adc, 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):

```python
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, -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).

19.2 Combining sources

gpiozero.tools.all_values(*values)

Returns the logical conjunction\(^{861}\) of all supplied values (the result is only True\(^{862}\) if and only if all input values are True).

---

861 https://en.wikipedia.org/wiki/Logical_conjunction
862 https://docs.python.org/3.5/library/constants.html#True
values are simultaneously True\(^{663}\). One or more values can be specified. For example, to light an LED (page 111) only when both buttons are pressed:

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

led = LED(4)
btn1 = Button(20)
btn2 = Button(21)

led.source = all_values(btn1, btn2)
pause()
```

gpiozero.tools.any_values(*values)

Returns the logical disjunction\(^{864}\) of all supplied values (the result is True\(^{865}\) if any of the input values are currently True\(^{866}\)). One or more values can be specified. For example, to light an LED (page 111) when any button is pressed:

```python
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, btn2)
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 (page 113) as the average of several potentiometers connected to an MCP3008 (page 135) ADC:

```python
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, pot2, pot3)
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 (page 113) as the product (i.e. multiplication) of several potentiometers connected to an MCP3008 (page 135) ADC:

```python
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, pot2, pot3)
pause()
```

(continues on next page)

---

\(^{663}\) https://docs.python.org/3.5/library/constants.html#True

\(^{864}\) https://en.wikipedia.org/wiki/Logical_disjunction

\(^{865}\) https://docs.python.org/3.5/library/constants.html#True

\(^{866}\) https://docs.python.org/3.5/library/constants.html#True

---

19.2. Combining sources 183
pot1 = MCP3008(channel=0)
pot2 = MCP3008(channel=1)
pot3 = MCP3008(channel=2)
led = PWMLED(4)

led.source = multiplied(pot1, pot2, pot3)
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:

```python
from gpiozero import MCP3008, PWMLED
from gpiozero.tools import summed, scaled
from signal import pause

pot1 = MCP3008(channel=0)
pot2 = MCP3008(channel=1)
pot3 = MCP3008(channel=2)
led = PWMLED(4)

led.source = scaled(summed(pot1, pot2, pot3), 0, 1, 0, 3)
pause()
```

gpiozero.tools.zip_values(*devices)
Provides a source constructed from the values of each item, for example:

```python
from gpiozero import MCP3008, Robot
from gpiozero.tools import zip_values
from signal import pause

robot = Robot(left=(4, 14), right=(17, 18))
left = MCP3008(0)
right = MCP3008(1)

robot.source = zip_values(left, right)
pause()

zip_values(left, right) is equivalent to zip(left.values, right.values).
```

### 19.3 Artificial sources

gpiozero.tools.alternating_values(initial_value=False)
Provides an infinite source of values alternating between True\(^{[67]}\) and False\(^{[68]}\), starting with initial_value (which defaults to False\(^{[69]}\)). For example, to produce a flashing LED:

```python
from gpiozero import LED
from gpiozero.tools import alternating_values
from signal import pause

(continues on next page)

---

\(^{[67]}\) [https://docs.python.org/3.5/library/constants.html#True](https://docs.python.org/3.5/library/constants.html#True)

\(^{[68]}\) [https://docs.python.org/3.5/library/constants.html#False](https://docs.python.org/3.5/library/constants.html#False)

\(^{[69]}\) [https://docs.python.org/3.5/library/constants.html#False](https://docs.python.org/3.5/library/constants.html#False)
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:

```python
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)
pause()
```

If you require a different range than -1 to +1, see scaled() (page 182).

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:

```python
from gpiozero import PWMLED
from gpiozero.tools import ramping_values
from signal import pause

red = PWMLED(2)

red.sourceDelay = 0.01
red.source = ramping_values(100)
pause()
```

If you require a wider range than 0 to 1, see scaled() (page 182).

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:

```python
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 182).

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:

```python
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)

pause()
```

If you require a different range than -1 to +1, see `scaled()` (page 182).
GPIO Zero includes a `Tone` class intended for use with the `TonalBuzzer` class. This class is in the `tones` module of GPIO Zero and is typically imported as follows:

```python
from gpiozero.tones import Tone
```

### 20.1 Tone

**class** `gpiozero.tones.Tone`

Represents a frequency of sound in a variety of musical notations.

The `Tone` class can be used with the `TonalBuzzer` class to easily represent musical tones. The class can be constructed in a variety of ways. For example as a straight frequency in Hz\(^{\text{70}}\) (which is the internal storage format), as an integer MIDI note, or as a string representation of a musical note.

All the following constructors are equivalent ways to construct the typical tuning note, concert A\(^{\text{71}}\) at 440Hz, which is MIDI note #69:

```python
>>> from gpiozero.tones import Tone
>>> Tone(440.0)
>>> Tone(69)
>>> Tone('A4')
```

If you do not want the constructor to guess which format you are using (there is some ambiguity between frequencies and MIDI notes at the bottom end of the frequencies, from 128Hz down), you can use one of the explicit constructors, `from_frequency()` (page 188), `from_midi()` (page 188), or `from_note()` (page 188), or you can specify a keyword argument when constructing:

```python
>>> Tone.from_frequency(440)
>>> Tone.from_midi(69)
>>> Tone.from_note('A4')
>>> Tone(frequency=440)
>>> Tone(midi=69)
>>> Tone(note='A4')
```

---

\(^{\text{70}}\) [https://en.wikipedia.org/wiki/Hertz](https://en.wikipedia.org/wiki/Hertz)

Several attributes are provided to permit conversion to any of the supported construction formats: \texttt{frequency} (page 188), \texttt{midi} (page 188), and \texttt{note} (page 188). Methods are provided to step \texttt{up()} (page 188) or \texttt{down()} (page 188) to adjacent MIDI notes.

\begin{footnotesize}
\begin{warning}
\hspace{1em} \textbf{Warning: } Currently \texttt{Tone} (page 187) derives from \texttt{float} and can be used as a floating point number in most circumstances (addition, subtraction, etc). This part of the API is not yet considered “stable”; i.e. we may decide to enhance / change this behaviour in future versions.
\end{warning}
\end{footnotesize}

\textbf{down}(n=1)
---
Return the \texttt{Tone} (page 187) \texttt{n} semi-tones below this frequency (\texttt{n} defaults to 1).

\textbf{classmethod from\_frequency}(freq)
---
Construct a \texttt{Tone} (page 187) from a frequency specified in Hz which must be a positive floating-point value in the range \(0 < \text{freq} \leq 20000\).

\textbf{classmethod from\_midi}(midi\_note)
---
Construct a \texttt{Tone} (page 187) from a MIDI note, which must be an integer in the range 0 to 127. For reference, A4 (concert A typically used for tuning) is MIDI note #69.

\textbf{classmethod from\_note}(note)
---
Construct a \texttt{Tone} (page 187) from a musical note which must consist of a capital letter A through G, followed by an optional semi-tone modifier (“b” for flat, “#” for sharp, or their Unicode equivalents), followed by an octave number (0 through 9).

For example concert A, the typical tuning note at 440Hz, would be represented as “A4”. One semi-tone above this would be “A#4” or alternatively “Bb4”. Unicode representations of sharp and flat are also accepted.

\textbf{up}(n=1)
---
Return the \texttt{Tone} (page 187) \texttt{n} semi-tones above this frequency (\texttt{n} defaults to 1).

\textbf{frequency}
---
Return the frequency of the tone in Hz.

\textbf{midi}
---
Return the (nearest) MIDI note to the tone’s frequency. This will be an integer number in the range 0 to 127. If the frequency is outside the range represented by MIDI notes (which is approximately 8Hz to 12.5KHz) \texttt{ValueError} exception will be raised.

\textbf{note}
---
Return the (nearest) note to the tone’s frequency. This will be a string in the form accepted by \texttt{from\_note()} (page 188). If the frequency is outside the range represented by this format (“A0” is approximately 27.5Hz, and “G9” is approximately 12.5KHz) a \texttt{ValueError} exception will be raised.

---

\footnotesize
\begin{footnotes}
\begin{itemize}
\item \texttt{https://docs.python.org/3.5/library/functions.html#float}
\item \texttt{https://en.wikipedia.org/wiki/Hertz}
\item \texttt{https://en.wikipedia.org/wiki/Concert_pitch}
\item \texttt{https://en.wikipedia.org/wiki/Concert_pitch}
\item \texttt{https://en.wikipedia.org/wiki/Hertz}
\item \texttt{https://docs.python.org/3.5/library/exceptions.html#ValueError}
\end{itemize}
\end{footnotes}

---
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:

### 21.1 pi_info

```python
gpiozero.pi_info(revision=None)
```

Returns a `PiBoardInfo` instance containing information about a revision of the Raspberry Pi.

**Parameters**

- `revision` *(str)*
  - The revision of the Pi to return information about. If this is omitted or None, the library will attempt to determine the model of Pi it is running on and return information about that.

### 21.2 PiBoardInfo

```python
class gpiozero.PiBoardInfo
```

This class is a named tuple\(^{881}\) 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:

```python
from gpiozero import *

print('{0}'.format(pi_info()))
print('{0:full}'.format(pi_info()))
print('{0:board}'.format(pi_info()))
print('{0:specs}'.format(pi_info()))
print('{0:headers}'.format(pi_info()))
```

“color” and “mono” can be prefixed to format specifications to force the use of ANSI color codes\(^{882}\). If

---

879 https://docs.python.org/3.5/library/stdtypes.html#str
880 https://docs.python.org/3.5/library/constants.html#None
881 https://docs.python.org/3.5/library/collections.html#collections.namedtuple
neither is specified, ANSI codes will only be used if stdout is detected to be a tty:

```python
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 215). If multiple pins support the desired function, `PinMultiplePins` (page 215) will be raised (use `physical_pins()` (page 190) if you expect multiple pins in the result, such as for electrical ground).

**Parameters function (str)** – The pin function you wish to search for. Usually this is something like “GPIO9” for Broadcom GPIO pin 9.

**physical_pins (function)**

Return the physical pins supporting the specified function as tuples of `(header, pin_number)` where `header` is a string specifying the header containing the `pin_number`. Note that the return value is a `set` which is not indexable. Use `physical_pin()` (page 190) if you are expecting a single return value.

**Parameters function (str)** – The pin function you wish to search for. Usually this is something like “GPIO9” for Broadcom GPIO pin 9, or “GND” for all the pins connecting to electrical ground.

**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 215) or `PinMultiplePins` (page 215) may be raised if the function is not associated with a single pin.

**Parameters function (str)** – The pin function you wish to determine pull-up for. Usually this is something like “GPIO9” for Broadcom GPIO pin 9.

**to_gpio (spec)**

Parses a pin `spec`, returning the equivalent Broadcom GPIO port number or raising a `ValueError` exception if the spec does not represent a GPIO port.

The `spec` may be given in any of the following forms:

- An integer, which will be accepted as a GPIO number
- ‘GPIOn’ where `n` is the GPIO number
- ‘WPIn’ where `n` is the `wiringPi` pin number
- ‘BCMn’ where `n` is the GPIO number (alias of GPIOn)
- ‘BOARDn’ where `n` is the physical pin number on the main header
- ‘h:n’ where `h` is the header name and `n` is the physical pin number (for example J8:5 is physical pin 5 on header J8, which is the main header on modern Raspberry Pis)
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\footnote{https://en.wikipedia.org/wiki/System_on_a_chip}) that this revision of the Pi is based upon.

manufacturer
A string indicating the name of the manufacturer (usually “Sony” but a few others exist).

memory
An integer indicating the amount of memory (in Mb) connected to the SoC.

Note: This can differ substantially from the amount of RAM available to the operating system as the GPU’s memory is shared with the CPU. When the camera module is activated, at least 128Mb of RAM is typically reserved for the GPU.

storage
A string indicating the type of bootable storage used with this revision of Pi, e.g. “SD”, “MicroSD”, or “eMMC” (for the Compute Module).

usb
An integer indicating how many USB ports are physically present on this revision of the Pi.

Note: This does not include the micro-USB port used to power the Pi.

ethernet
An integer indicating how many Ethernet ports are physically present on this revision of the Pi.

wifi
A bool indicating whether this revision of the Pi has wifi built-in.

bluetooth
A bool indicating whether this revision of the Pi has bluetooth built-in.

csi
An integer indicating the number of CSI (camera) ports available on this revision of the Pi.

dsi
An integer indicating the number of DSI (display) ports available on this revision of the Pi.

headers
A dictionary which maps header labels to \texttt{HeaderInfo} (page 192) tuples. For example, to obtain information about header P1 you would query \texttt{headers['P1']}. To obtain information about pin 12 on header J8 you would query \texttt{headers['J8'].pins[12]}.

\footnote{https://en.wikipedia.org/wiki/System_on_a_chip}
A rendered version of this data can be obtained by using the `PiBoardInfo` (page 189) object in a format string:

```python
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 189) object in a format string. For example:

```python
from gpiozero import *
print('{0:board}'.format(pi_info()))
```

## 21.3 HeaderInfo

**class gpiozero.HeaderInfo**

This class is a `collections.namedtuple()` 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:

```python
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']))
```

“color” and “mono” can be prefixed to format specifications to force the use of ANSI color codes. If neither is specified, ANSI codes will only be used if stdout is detected to be a tty:

```python
print('{0:color row2}'.format(pi_info().headers['J8']))  # force use of ANSI colors
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 193) tuples.

---

893 https://docs.python.org/3.5/library/collections.html#collections.namedtuple
895 https://docs.python.org/3.5/library/constants.html#None
896 https://docs.python.org/3.5/library/constants.html#True
897 https://docs.python.org/3.5/library/constants.html#False
21.4 PinInfo

class gpiozero.PinInfo

This class is a \texttt{namedtuple()} derivative used to represent information about a pin present on a GPIO header. The following attributes are defined:

\textbf{number}

An integer containing the physical pin number on the header (starting from 1 in accordance with convention).

\textbf{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.

\textbf{pull_up}

A bool indicating whether the pin has a physical pull-up resistor permanently attached (this is usually \texttt{False} but GPIO2 and GPIO3 are usually \texttt{True}). This is used internally by gpiozero to raise errors when pull-down is requested on a pin with a physical pull-up resistor.

\textbf{row}

An integer indicating on which row the pin is physically located in the header (1-based)

\textbf{col}

An integer indicating in which column the pin is physically located in the header (1-based)

\footnote{https://docs.python.org/3.5/library/collections.html#collections.namedtuple}
\footnote{https://docs.python.org/3.5/library/constants.html#False}
\footnote{https://docs.python.org/3.5/library/constants.html#True}
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 198) which turns it into a Pin (page 199) implementation. The default factory can be queried (and changed) with Device.pin_factory (page 175). However, all classes (even internal devices) 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 in the Configuring Remote GPIO (page 43) 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` will be raised.

### 22.1 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 198) and `Pin` (page 199) classes are listed in the table below. Factories are listed in the order that they are tried by default.

<table>
<thead>
<tr>
<th>Name</th>
<th>Factory class</th>
<th>Pin class</th>
</tr>
</thead>
<tbody>
<tr>
<td>rpigpio</td>
<td>gpiozero.pins.rpigpio.RPiGPIOFactory (page 207)</td>
<td>gpiozero.pins.rpigpio.RPiGPIOPin (page 207)</td>
</tr>
<tr>
<td>rpio</td>
<td>gpiozero.pins.rpio.RPIOFactory (page 207)</td>
<td>gpiozero.pins.rpio.RPIOPin (page 207)</td>
</tr>
<tr>
<td>pigpio</td>
<td>gpiozero.pins.pigpio.PiGPIOFactory (page 207)</td>
<td>gpiozero.pins.pigpio.PiGPIOPin (page 208)</td>
</tr>
<tr>
<td>native</td>
<td>gpiozero.pins.native.NativeFactory (page 208)</td>
<td>gpiozero.pins.native.NativePin (page 208)</td>
</tr>
</tbody>
</table>

If you need to change the default pin factory from within a script, either set `Device.pin_factory` (page 175) to the new factory instance to use:

---

901 [https://docs.python.org/3.5/library/exceptions.html#ImportError](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 208) on a remote pi called “remote-pi” you can set the
PIGPIO_ADDR (page 74) environment variable when running your script:

$ GPIOZERO_PIN_FACTORY=pigpio PIGPIO_ADDR=remote-pi python3 my_script.py

Like the GPIOZERO_PIN_FACTORY (page 74) 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
RPiGPIOPin (page 207) for one pin, and NativeFactory (page 208) for another. This is un-
supported, 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 43).

22.2 Mock pins

There’s also a gpiozero.pins.mock.MockFactory (page 209) 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

(continues on next page)
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.

## 22.3 Base classes

```python
# 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 electrically 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)
```

### `gpiozero.Factory`

**Generates pins and SPI interfaces for devices. This is an abstract base class for pin factories. Descendents must override the following methods:**

- `ticks()` (page 199)
- `ticks_diff()` (page 199)

**Descendents may override the following methods, if applicable:**

- `close()` (page 198)
- `reserve_pins()` (page 199)
- `release_pins()` (page 198)
- `release_all()` (page 198)
- `pin()` (page 198)
- `spi()` (page 199)

#### `close()`

Closes the pin factory. This is expected to clean up all resources manipulated by the factory. It it typically called at script termination.

#### `pin(spec)`

Creates an instance of a Pin (page 199) descendant 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 198) 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 198) for further information.
release_pins (reserver, *pins)
Releases the reservation of reserver against pins. This is typically called during close() (page 175) 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 198).

spi (**spi_args)
Returns an instance of an SPI (page 202) 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 213).

ticks()
Return the current ticks, according to the factory. The reference point is undefined and thus the result of this method is only meaningful when compared to another value returned by this method.

The format of the time is also arbitrary, as is whether the time wraps after a certain duration. Ticks should only be compared using the ticks_diff() (page 199) method.

ticks_diff (later, earlier)
Return the time in seconds between two ticks() (page 199) results. The arguments are specified in the same order as they would be in the formula later - earlier but the result is guaranteed to be in seconds, and to be positive even if the ticks “wrapped” between calls to ticks() (page 199).

pi_info
Returns a PiBoardInfo (page 189) 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 200)
• output_with_state() (page 200)
• input_with_pull() (page 200)
• _set_state()
• _get_frequency()
• _set_frequency()
• _get_pull()
• _set_pull()
• _get_bounce()
• _set_bounce()
• _get_edges()

902 https://docs.python.org/3.5/library/constants.html#None
• `_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 199) 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:

```python
pin.function = 'input'
pin.pull = pull
```

However, descendents may override this order to provide the smallest possible delay between configuring the pin for input and pulling the pin up/down (which can be important for avoiding “blips” in some configurations).

`output_with_state (state)`
Sets the pin’s function to “output” and specifies an initial state for the pin. By default this is equivalent to performing:

```python
pin.function = 'output'
pin.state = state
```

However, descendents may override this in order to provide the smallest possible delay between configuring the pin for output and specifying an initial value (which can be important for avoiding “blips” in active-low configurations).

`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 200) is currently “rising”, `bounce` (page 200) is currently 5/1000 (5ms), then the waveform below will only fire `when_changed` (page 201) on two occasions despite there being three rising edges:

<table>
<thead>
<tr>
<th>TIME 0...1...2...3...4...5...6...7...8...9...10...11...12 ms</th>
</tr>
</thead>
<tbody>
<tr>
<td>bounce elimination</td>
</tr>
<tr>
<td>HIGH = = = = &gt;</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>LOW = = = = = = = =</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>when_changed fires</td>
</tr>
</tbody>
</table>

If the pin does not support edge detection, attempts to set this property will raise `PinEdgeDetectUnsupported` (page 214). 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 201). This can be one of the strings “both” (the default), “rising”, “falling”, or “none”:

903 https://docs.python.org/3.5/library/constants.html#None
If the pin does not support edge detection, attempts to set this property will raise `PinEdgeDetectUnsupported` (page 214).

**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 215) 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 214) 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 214). If the specified value is not supported by the underlying hardware, `PinInvalidPull` (page 214) 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 201) for more information):

If a pin is currently configured for input, and an attempt is made to set this attribute, `PinSetInput` (page 214) will be raised. If an invalid value is specified for this attribute, `PinInvalidState` (page 214) 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 200) is detected on the pin). The function or bound method must accept two parameters: the first will report the ticks (from `Factory.ticks()` (page 199)) when the pin’s state changed, and the second will report the pin’s current state.

---

*https://docs.python.org/3.5/library/constants.html#None
https://docs.python.org/3.5/library/constants.html#None
https://docs.python.org/3.5/library/constants.html#None
https://docs.python.org/3.5/library/constants.html#None
https://docs.python.org/3.5/library/constants.html#None

22.3. Base classes 201
Warning: Depending on hardware support, the state is not guaranteed to be accurate. For instance, many GPIO implementations will provide an interrupt indicating when a pin’s state changed but not what it changed to. In this case the pin driver simply reads the pin’s current state to supply this parameter, but the pin’s state may have changed since the interrupt. Exercise appropriate caution when relying upon this parameter.

If the pin does not support edge detection, attempts to set this property will raise `PinEdgeDetectUnsupported` (page 214).

class gpiozero.SPI

Abstract interface for Serial Peripheral Interface\(^9\) (SPI) implementations. Descendents must override the following methods:

- `transfer()` (page 202)
- `_get_clock_mode()`

Descendents may override the following methods, if applicable:

- `read()` (page 202)
- `write()` (page 202)
- `_set_clock_mode()`
- `_get_lsb_first()`
- `_set_lsb_first()`
- `_get_select_high()`
- `_set_select_high()`
- `_get_bits_per_word()`
- `_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 202) of the interface.

This method is typically used with read-only devices that feature half-duplex communication. See `transfer()` (page 202) 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 202) 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 202) 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 202) 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 202) 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 203) and `clock_phase` (page 203) attributes combined according to the following table:

<table>
<thead>
<tr>
<th>mode</th>
<th>polarity (CPOL)</th>
<th>phase (CPHA)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>1</td>
<td>False</td>
<td>True</td>
</tr>
<tr>
<td>2</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>3</td>
<td>True</td>
<td>True</td>
</tr>
</tbody>
</table>

Adjusting this value adjusts both the `clock_polarity` (page 203) and `clock_phase` (page 203) 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 203) attribute (corresponding to CPOL).

The following diagram indicates when data is read when `clock_polarity` (page 203) is `False`, and `clock_phase` (page 203) is `False` (the default), equivalent to CPHA 0:

![Diagram of CPHA 0](https://example.com/diagram.png)

The following diagram indicates when data is read when `clock_polarity` (page 203) is `False`, but `clock_phase` (page 203) is `True`, equivalent to CPHA 1:

![Diagram of CPHA 1](https://example.com/diagram.png)

**clock_polarity**

The polarity of the SPI clock pin. If this is `False` (the default), the clock pin willidle 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.

---

909 https://docs.python.org/3.5/library/constants.html#False
910 https://docs.python.org/3.5/library/constants.html#True
911 https://docs.python.org/3.5/library/constants.html#False
912 https://docs.python.org/3.5/library/constants.html#False
913 https://docs.python.org/3.5/library/constants.html#False
914 https://docs.python.org/3.5/library/constants.html#False
915 https://docs.python.org/3.5/library/constants.html#False
916 https://docs.python.org/3.5/library/constants.html#True
The following diagram illustrates the waveform when `clock_polarity` (page 203) is `False` (the default), equivalent to CPOL 0:

![Waveform Diagram 1](image1)

The following diagram illustrates the waveform when `clock_polarity` (page 203) is `True`, equivalent to CPOL 1:

![Waveform Diagram 2](image2)

**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 of the connection.

The following diagram shows a word containing the number 5 (binary 0101) transmitted on MISO with `bits_per_word` (page 202) set to 4, and `clock_mode` (page 203) set to 0, when `lsb_first` (page 204) is `False` (the default):

![Waveform Diagram 3](image3)

And now with `lsb_first` (page 204) set to `True` (and all other parameters the same):

![Waveform Diagram 4](image4)

**select_high**

If `False` (the default), the chip select line is considered active when it is pulled low. When set to

---

917 https://docs.python.org/3.5/library/constants.html#False
918 https://docs.python.org/3.5/library/constants.html#True
919 https://docs.python.org/3.5/library/constants.html#False
920 https://en.wikipedia.org/wiki/Endianness#Bit_endianness
921 https://docs.python.org/3.5/library/constants.html#False
922 https://docs.python.org/3.5/library/constants.html#True
923 https://docs.python.org/3.5/library/constants.html#False
True\textsuperscript{924}, 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 \texttt{clock_polarity} (page 203) is \texttt{False}\textsuperscript{925}, and \texttt{select_high} (page 204) is \texttt{False}\textsuperscript{926} (the default):

![Waveform Diagram]

And when \texttt{select_high} (page 204) is \texttt{True}\textsuperscript{927}:

![Waveform Diagram]

\texttt{class gpiozero.pins.pi.PiFactory}

Extends \texttt{Factory} (page 198). Abstract base class representing hardware attached to a Raspberry Pi. This forms the base of \texttt{LocalPiFactory} (page 206).

\texttt{close()}

Closes the pin factory. This is expected to clean up all resources manipulated by the factory. It it typically called at script termination.

\texttt{pin(spec)}

Creates an instance of a \texttt{Pin} descendent representing the specified pin.

\textbf{Warning:} Descendents must ensure that pin instances representing the same hardware are identical; i.e. two separate invocations of \texttt{pin()} (page 205) for the same pin specification must return the same object.

\texttt{release_pins(reserver, *pins)}

Releases the reservation of \texttt{reservation} against \texttt{pins}. This is typically called during \texttt{close()} (page 175) 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).

\texttt{reserve_pins(requester, *pins)}

Called to indicate that the device reserves the right to use the specified \texttt{pins}. This should be done during device construction. If pins are reserved, you must ensure that the reservation is released by eventually called \texttt{release_pins()} (page 205).

\texttt{spi(**spi_args)}

Returns an SPI interface, for the specified SPI \texttt{port} and \texttt{device}, or for the specified pins (\texttt{clock_pin}, \texttt{mosi_pin}, \texttt{miso_pin}, and \texttt{select_pin}). Only one of the schemes can be used; attempting to mix \texttt{port} and \texttt{device} with pin numbers will raise \texttt{SPIBadArgs} (page 213).

\textsuperscript{924} https://docs.python.org/3.5/library/constants.html\#True
\textsuperscript{925} https://docs.python.org/3.5/library/constants.html\#False
\textsuperscript{926} https://docs.python.org/3.5/library/constants.html\#False
\textsuperscript{927} https://docs.python.org/3.5/library/constants.html\#True
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 hardware based interface (using spidev) will be returned. Otherwise, a software based interface 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).

```python
class gpiozero.pins.pi.PiPin(factory, number)
```
Extends `Pin` (page 199). Abstract base class representing a multi-function GPIO pin attached to a Raspberry Pi. 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()`

```python
class gpiozero.pins.local.LocalPiFactory
```
Extends `PiFactory` (page 205). Abstract base class representing pins attached locally to a Pi. This forms the base class for local-only pin interfaces (`RPiGPIOPin` (page 207), `RPIOPin` (page 207), and `NativePin` (page 208)).

```python
static ticks()
```
Return the current ticks, according to the factory. The reference point is undefined and thus the result of this method is only meaningful when compared to another value returned by this method.

The format of the time is also arbitrary, as is whether the time wraps after a certain duration. Ticks should only be compared using the `ticks_diff()` (page 206) method.

```python
static ticks_diff(later, earlier)
```
Return the time in seconds between two `ticks()` (page 206) results. The arguments are specified in the same order as they would be in the formula `later - earlier` but the result is guaranteed to be in seconds, and to be positive even if the ticks “wrapped” between calls to `ticks()` (page 206).

```python
class gpiozero.pins.local.LocalPiPin(factory, number)
```
Extends `PiPin` (page 206). Abstract base class representing a multi-function GPIO pin attached to the local Raspberry Pi.
22.4 RPi.GPIO

```python
class gpiozero.pins.rpigpio.RPiGPIOFactory
    Extends LocalPiFactory (page 206). Uses the RPi.GPIO\(^{928}\) library to interface to the Pi’s GPIO pins. This is the default pin implementation if the RPi.GPIO library is installed. Supports all features including PWM (via software).

Because this is the default pin implementation you can use it simply by specifying an integer number for the pin in most operations, e.g.:

```python
from gpiozero import LED
led = LED(12)
```

However, you can also construct RPi.GPIO pins manually if you wish:

```python
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)
    Extends LocalPiPin (page 206). Pin implementation for the RPi.GPIO\(^{929}\) library. See RPiGPIOFactory (page 207) for more information.
```

22.5 RPIO

```python
class gpiozero.pins.rpio.RPIOFactory
    Extends LocalPiFactory (page 206). Uses the RPIO\(^{930}\) 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:

```python
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)
    Extends LocalPiPin (page 206). Pin implementation for the RPIO\(^{931}\) library. See RPIOFactory (page 207) for more information.
```

22.6 PiGPIO

```python
class gpiozero.pins.pigpio.PiGPIOFactory (host=None, port=None)
    Extends PiFactory (page 205). Uses the ‘pigpio’\(^{932}\) library to interface to the Pi’s GPIO pins. The
```
\(^{928}\) https://pypi.python.org/pypi/RPi.GPIO
\(^{929}\) https://pypi.python.org/pypi/RPi.GPIO
\(^{930}\) https://pythonhosted.org/RPIO/
\(^{931}\) https://pythonhosted.org/RPIO/
\(^{932}\) https://pypi.python.org/pypi/piGPIO

Gpiozero Documentation, Release 1.5.1

pigpio library relies on a daemon (pigpiod) to be running as root to provide access to the GPIO pins, and communicates with this daemon over a network socket.

While this does mean only the daemon itself should control the pins, the architecture does have several advantages:

- Pins can be remote controlled from another machine (the other machine doesn’t even have to be a Raspberry Pi; it simply needs the `pigpio` client library installed on it)
- The daemon supports hardware PWM via the DMA controller
- Your script itself doesn’t require root privileges; it just needs to be able to communicate with the daemon

You can construct pigpio pins manually like so:

```python
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:

```python
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)

Extends PiPin (page 206). Pin implementation for the `pigpio` library. See PiGPIOFactory (page 207) for more information.

**22.7 Native**

class gpiozero.pins.native.NativeFactory

Extends LocalPiFactory (page 206). 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.

You can construct native pin instances manually like so:

```python
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)

Extends LocalPiPin (page 206). Native pin implementation. See NativeFactory (page 208) for more information.

22.8 Mock

class gpiozero.pins.mock.MockFactory (revision=None, pin_class=None)

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 (page 199) 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 209) method by default. This can be changed after construction by modifying the pin_class (page 209) attribute.

pin_class

This attribute stores the MockPin (page 209) class (or descendent) that will be used when constructing pins with the pin() (page 209) method (if no pin_class parameter is used to override it). It defaults on construction to the value of the pin_class parameter in the constructor, or MockPin (page 209) if that is unspecified.

pin(spec, pin_class=None, **kwargs)

The pin method for MockFactory (page 209) additionally takes a pin_class attribute which can be used to override the class’ pin_class (page 209) attribute. Any additional keyword arguments will be passed along to the pin constructor (useful with things like MockConnectedPin (page 209) 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.

class gpiozero.pins.mock.MockPWMPin (factory, number)

This derivative of MockPin (page 209) adds PWM support.

class gpiozero.pins.mock.MockConnectedPin (factory, number, input_pin=None)

This derivative of MockPin (page 209) 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 209) 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 209) is intended to be used with another MockPin (page 209) 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.
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:

```python
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 211), 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 130) you would expect a `ValueError` to be raised. In fact, a `OutputDeviceBadValue` (page 214) error will be raised. However, note that this descends from both `GPIOZeroError` (page 211) (indirectly) and from `ValueError` so you can still do the obvious:

```python
from gpiozero import *
led = PWMLED(17)
try:
    led.value = 2
except ValueError:
    print('Bad value specified')
```

### 23.1 Errors

```
exception gpiozero.GPIOZeroError
    Bases: Exception

Base class for all exceptions in GPIO Zero
```

932 https://docs.python.org/3.5/library/exceptions.html#ValueError  
933 https://docs.python.org/3.5/library/exceptions.html#ValueError  
934 https://docs.python.org/3.5/library/exceptions.html#Exception
exception gpiozero.DeviceClosed
    Bases: gpiozero.exc.GPIOZeroError
    Error raised when an operation is attempted on a closed device

exception gpiozero.BadEventHandler
    Bases: gpiozero.exc.GPIOZeroError, ValueError
    Error raised when an event handler with an incompatible prototype is specified

exception gpiozero.BadWaitTime
    Bases: gpiozero.exc.GPIOZeroError, ValueError
    Error raised when an invalid wait time is specified

exception gpiozero.BadQueueLen
    Bases: gpiozero.exc.GPIOZeroError, ValueError
    Error raised when non-positive queue length is specified

exception gpiozero.BadPinFactory
    Bases: gpiozero.exc.GPIOZeroError, ImportError
    Error raised when an unknown pin factory name is specified

exception gpiozero.ZombieThread
    Bases: gpiozero.exc.GPIOZeroError, RuntimeError
    Error raised when a thread fails to die within a given timeout

exception gpiozero.CompositeDeviceError
    Bases: gpiozero.exc.GPIOZeroError
    Base class for errors specific to the CompositeDevice hierarchy

exception gpiozero.CompositeDeviceBadName
    Bases: gpiozero.exc.CompositeDeviceError, ValueError
    Error raised when a composite device is constructed with a reserved name

exception gpiozero.CompositeDeviceBadOrder
    Bases: gpiozero.exc.CompositeDeviceError, ValueError
    Error raised when a composite device is constructed with an incomplete order

exception gpiozero.CompositeDeviceBadDevice
    Bases: gpiozero.exc.CompositeDeviceError, ValueError
    Error raised when a composite device is constructed with an object that doesn’t inherit from Device (page 175)

exception gpiozero.EnergenieSocketMissing
    Bases: gpiozero.exc.CompositeDeviceError, ValueError
    Error raised when socket number is not specified

exception gpiozero.EnergenieBadSocket
    Bases: gpiozero.exc.CompositeDeviceError, ValueError
    Error raised when an invalid socket number is passed to Energenie (page 159)
exception gpiozero.SPIError
   Bases: gpiozero.exc.GPIOZeroError
   Base class for errors related to the SPI implementation

exception gpiozero.SPIBadArgs
   Bases: gpiozero.exc.SPIError, ValueError
   Error raised when invalid arguments are given while constructing SPIDevice (page 139)

exception gpiozero.SPIBadChannel
   Bases: gpiozero.exc.SPIError, ValueError
   Error raised when an invalid channel is given to an AnalogInputDevice (page 139)

exception gpiozero.SPIFixedClockMode
   Bases: gpiozero.exc.SPIError, AttributeError
   Error raised when the SPI clock mode cannot be changed

exception gpiozero.SPIInvalidClockMode
   Bases: gpiozero.exc.SPIError, ValueError
   Error raised when an invalid clock mode is given to an SPI implementation

exception gpiozero.SPIFixedBitOrder
   Bases: gpiozero.exc.SPIError, AttributeError
   Error raised when the SPI bit-endianness cannot be changed

exception gpiozero.SPIFixedSelect
   Bases: gpiozero.exc.SPIError, AttributeError
   Error raised when the SPI select polarity cannot be changed

exception gpiozero.SPIFixedWordSize
   Bases: gpiozero.exc.SPIError, AttributeError
   Error raised when the number of bits per word cannot be changed

exception gpiozero.SPIInvalidWordSize
   Bases: gpiozero.exc.SPIError, ValueError
   Error raised when an invalid (out of range) number of bits per word is specified

exception gpiozero.GPIODeviceError
   Bases: gpiozero.exc.GPIOZeroError
   Base class for errors specific to the GPIODevice hierarchy

exception gpiozero.GPIODeviceClosed
   Bases: gpiozero.exc.GPIODeviceError, gpiozero.exc.DeviceClosed
   Deprecated descendent of DeviceClosed (page 211)

exception gpiozero.GPIOPinInUse
   Bases: gpiozero.exc.GPIODeviceError
   Error raised when attempting to use a pin already in use by another device

exception gpiozero.GPIOPinMissing
   Bases: gpiozero.exc.GPIODeviceError, ValueError
   945 https://docs.python.org/3.5/library/exceptions.html#ValueError
   946 https://docs.python.org/3.5/library/exceptions.html#ValueError
   947 https://docs.python.org/3.5/library/exceptions.html#AttributeError
   948 https://docs.python.org/3.5/library/exceptions.html#ValueError
   949 https://docs.python.org/3.5/library/exceptions.html#AttributeError
   950 https://docs.python.org/3.5/library/exceptions.html#AttributeError
   951 https://docs.python.org/3.5/library/exceptions.html#AttributeError
   952 https://docs.python.org/3.5/library/exceptions.html#ValueError
   953 https://docs.python.org/3.5/library/exceptions.html#ValueError

23.1. Errors 213
Error raised when a pin specification is not given

```python
exception gpiozero.InputDeviceError
    Bases: gpiozero.exc.GPIODeviceError
    Base class for errors specific to the InputDevice hierarchy
```

```python
exception gpiozero.OutputDeviceError
    Bases: gpiozero.exc.GPIODeviceError
    Base class for errors specified to the OutputDevice hierarchy
```

```python
exception gpiozero.OutputDeviceBadValue
    Bases: gpiozero.exc.OutputDeviceError, ValueError
    Error raised when value is set to an invalid value
```

```python
exception gpiozero.PinError
    Bases: gpiozero.exc.GPIOZeroError
    Base class for errors related to pin implementations
```

```python
exception gpiozero.PinInvalidFunction
    Bases: gpiozero.exc.PinError, ValueError
    Error raised when attempting to change the function of a pin to an invalid value
```

```python
exception gpiozero.PinInvalidState
    Bases: gpiozero.exc.PinError, ValueError
    Error raised when attempting to assign an invalid state to a pin
```

```python
exception gpiozero.PinInvalidPull
    Bases: gpiozero.exc.PinError, ValueError
    Error raised when attempting to assign an invalid pull-up to a pin
```

```python
exception gpiozero.PinInvalidEdges
    Bases: gpiozero.exc.PinError, ValueError
    Error raised when attempting to assign an invalid edge detection to a pin
```

```python
exception gpiozero.PinInvalidBounce
    Bases: gpiozero.exc.PinError, ValueError
    Error raised when attempting to assign an invalid bounce time to a pin
```

```python
exception gpiozero.PinSetInput
    Bases: gpiozero.exc.PinError, AttributeError
    Error raised when attempting to set a read-only pin
```

```python
exception gpiozero.PinFixedPull
    Bases: gpiozero.exc.PinError, AttributeError
    Error raised when attempting to set the pull of a pin with fixed pull-up
```

```python
exception gpiozero.PinEdgeDetectUnsupported
    Bases: gpiozero.exc.PinError, AttributeError
    Error raised when attempting to use edge detection on unsupported pins
```

---

954 https://docs.python.org/3.5/library/exceptions.html#ValueError
955 https://docs.python.org/3.5/library/exceptions.html#ValueError
956 https://docs.python.org/3.5/library/exceptions.html#ValueError
957 https://docs.python.org/3.5/library/exceptions.html#ValueError
958 https://docs.python.org/3.5/library/exceptions.html#ValueError
959 https://docs.python.org/3.5/library/exceptions.html#ValueError
960 https://docs.python.org/3.5/library/exceptions.html#AttributeError
961 https://docs.python.org/3.5/library/exceptions.html#AttributeError
962 https://docs.python.org/3.5/library/exceptions.html#AttributeError
exception gpiozero.PinUnsupported
   Bases: gpiozero.exc.PinError, NotImplementedError
         Error raised when attempting to obtain a pin interface on unsupported pins

exception gpiozero.PinSPIUnsupported
   Bases: gpiozero.exc.PinError, NotImplementedError
         Error raised when attempting to obtain an SPI interface on unsupported pins

exception gpiozero.PinPWMError
   Bases: gpiozero.exc.PinPWMError
         Base class for errors related to PWM implementations

exception gpiozero.PinPWMUnsupported
   Bases: gpiozero.exc.PinPWMError, AttributeError
         Error raised when attempting to activate PWM on unsupported pins

exception gpiozero.PinPWMFixedValue
   Bases: gpiozero.exc.PinPWMError, AttributeError
         Error raised when attempting to initialize PWM on an input pin

exception gpiozero.PinUnknownPi
   Bases: gpiozero.exc.PinError, RuntimeError
         Error raised when gpiozero doesn’t recognize a revision of the Pi

exception gpiozero.PinMultiplePins
   Bases: gpiozero.exc.PinError, RuntimeError
         Error raised when multiple pins support the requested function

exception gpiozero.PinNoPins
   Bases: gpiozero.exc.PinError, RuntimeError
         Error raised when no pins support the requested function

exception gpiozero.PinInvalidPin
   Bases: gpiozero.exc.PinError, ValueError
         Error raised when an invalid pin specification is provided

23.2 Warnings

exception gpiozero.GPIOZeroWarning
   Bases: Warning
         Base class for all warnings in GPIO Zero

exception gpiozero.DistanceSensorNoEcho
   Bases: gpiozero.exc.GPIOZeroWarning
         Warning raised when the distance sensor sees no echo at all

---

963 https://docs.python.org/3.5/library/exceptions.html#NotImplementedError
964 https://docs.python.org/3.5/library/exceptions.html#NotImplementedError
965 https://docs.python.org/3.5/library/exceptions.html#AttributeError
966 https://docs.python.org/3.5/library/exceptions.html#AttributeError
967 https://docs.python.org/3.5/library/exceptions.html#RuntimeError
968 https://docs.python.org/3.5/library/exceptions.html#RuntimeError
969 https://docs.python.org/3.5/library/exceptions.html#RuntimeError
970 https://docs.python.org/3.5/library/exceptions.html#ValueError
971 https://docs.python.org/3.5/library/exceptions.html#Warning

---

23.2. Warnings 215
exception gpiozero.SPIWarning
    Bases: gpiozero.exc.GPIOZeroWarning
    Base class for warnings related to the SPI implementation

exception gpiozero.SPISoftwareFallback
    Bases: gpiozero.exc.SPIWarning
    Warning raised when falling back to the software implementation

exception gpiozero.PinWarning
    Bases: gpiozero.exc.GPIOZeroWarning
    Base class for warnings related to pin implementations

exception gpiozero.PinFactoryFallback
    Bases: gpiozero.exc.PinWarning
    Warning raised when a default pin factory fails to load and a fallback is tried

exception gpiozero.PinNonPhysical
    Bases: gpiozero.exc.PinWarning
    Warning raised when a non-physical pin is specified in a constructor

exception gpiozero.ThresholdOutOfRange
    Bases: gpiozero.exc.GPIOZeroWarning
    Warning raised when a threshold is out of range specified by min and max values

exception gpiozero.CallbackSetToNone
    Bases: gpiozero.exc.GPIOZeroWarning
    Warning raised when a callback is set to None when its previous value was None
24.1 Release 1.5.1 (2019-06-24)

* Added Raspberry Pi 4 data for `pi_info()` (page 189) and `pinout` * Minor docs updates

24.2 Release 1.5.0 (2019-02-12)

- Introduced pin event timing to increase accuracy of certain devices such as the HC-SR04 DistanceSensor (page 101). (#664972, #665973)
- Further improvements to DistanceSensor (page 101) (ignoring missed edges). (#719974)
- Allow source to take a device object as well as values or other values. See `Source/Values` (page 59). (#640975)
- Added internal device classes LoadAverage (page 170) and DiskUsage (page 171) (thanks to Jeevan M R for the latter). (#532976, #714977)
- Added support for colorzero978 with RGBLED (page 115) (this adds a new dependency). (#655979)
- Added TonalBuzzer (page 119) with Tone (page 187) API for specifying frequencies raw or via MIDI or musical notes. (#681980, #717981)
- Added PiHutXmasTree (page 149). (#502982)

972 https://github.com/RPi-Distro/python-gpiozero/issues/664
973 https://github.com/RPi-Distro/python-gpiozero/issues/665
974 https://github.com/RPi-Distro/python-gpiozero/issues/719
975 https://github.com/RPi-Distro/python-gpiozero/issues/640
976 https://github.com/RPi-Distro/python-gpiozero/issues/532
977 https://github.com/RPi-Distro/python-gpiozero/issues/714
979 https://github.com/RPi-Distro/python-gpiozero/issues/655
980 https://github.com/RPi-Distro/python-gpiozero/issues/681
981 https://github.com/RPi-Distro/python-gpiozero/issues/717
982 https://github.com/RPi-Distro/python-gpiozero/issues/502
• Added *PumpkinPi* (page 162) and *JamHat* (page 154) (thanks to Claire Pollard). (#680983, #681984, #717985)

• Ensured gpiozero can be imported without a valid pin factory set. (#591986, #713987)

• Reduced import time by not computing default pin factory at the point of import. (#675988, #722989)

• Added support for various pin numbering mechanisms. (#470990)

• *Motor* (page 120) instances now use *DigitalOutputDevice* (page 127) for non-PWM pins.

• Allow non-PWM use of *Robot* (page 155). (#481991)

• Added optional `enable init` param to *Motor* (page 120). (#366992)

• Added `--xyz` option to `pinout` command line tool to open `pinout.xyz` in a web browser. (#604994)

• Added 3B+, 3A+ and CM3+ to Pi model data. (#627995, #704996)

• Minor improvements to *Energenie* (page 159), thanks to Steve Amor. (#629997, #634998)

• Allow *SmoothedInputDevice* (page 105), *LightSensor* (page 99) and *MotionSensor* (page 97) to have pull-up configured. (#652999)

• Allow input devices to be pulled up or down externally, thanks to Philippe Muller. (#5931000, #6581001)

• Minor changes to support Python 3.7, thanks to Russel Winder and Rick Ansell. (#6661002, #6681003, #6691004, #6711005, #6731006)

• Added `zip_values()` (page 184) source tool.

• Correct row/col numbering logic in *PinInfo* (page 193). (#6741007)

• Many additional tests, and other improvements to the test suite.

• Many documentation corrections, additions and clarifications.

• Automatic documentation class hierarchy diagram generation.

• Automatic copyright attribution in source files.

### 24.3 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 156) and `Robot.backward()` (page 155). (#306 and #619)

• Fixed `DistanceSensor` (page 101) returning incorrect readings after a long pause, and added a lock to ensure multiple distance sensors can operate simultaneously in a single project. (#584, #595, #617, #618)

• Added support for phase/enable motor drivers with `PhaseEnableMotor` (page 122), `PhaseEnableRobot` (page 156), and descendants, thanks to Ian Harcombe! (#386)

• A variety of other minor enhancements, largely thanks to Andrew Scheller! (#479, #489, #491, #492)

24.4 Release 1.4.0 (2017-07-26)

• Pin factory is now configurable from device constructors (page 196) 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 assignations (#279)

• Major work on SPI, primarily to support remote hardware SPI (#421, #459, #465, #468, #575)

• Pin reservation now works properly between GPIO and SPI devices (#459, #468)

• Lots of work on the documentation: `source/values` chapter (page 59), better charts, more recipes, remote GPIO configuration (page 43), mock pins, better PDF output (#484, #469, #523, #520, #434, #565, #576)

• Support for `StatusZero` (page 160) and `StatusBoard` (page 160) HATs (#558)

• Added `pinout` command line tool to provide a simple reference to the GPIO layout and information about the associated Pi (#497, #504) thanks to Stewart Adcock for the initial work

• `pi_info()` (page 189) made more lenient for new (unknown) Pi models (#529)

---

1008 https://github.com/RPi-Distro/python-gpiozero/issues/306
1009 https://github.com/RPi-Distro/python-gpiozero/issues/619
1010 https://github.com/RPi-Distro/python-gpiozero/issues/584
1011 https://github.com/RPi-Distro/python-gpiozero/issues/595
1012 https://github.com/RPi-Distro/python-gpiozero/issues/617
1013 https://github.com/RPi-Distro/python-gpiozero/issues/618
1014 https://github.com/RPi-Distro/python-gpiozero/issues/386
1015 https://github.com/RPi-Distro/python-gpiozero/issues/479
1016 https://github.com/RPi-Distro/python-gpiozero/issues/489
1017 https://github.com/RPi-Distro/python-gpiozero/issues/491
1018 https://github.com/RPi-Distro/python-gpiozero/issues/492
1019 https://github.com/RPi-Distro/python-gpiozero/issues/279
1020 https://github.com/RPi-Distro/python-gpiozero/issues/421
1021 https://github.com/RPi-Distro/python-gpiozero/issues/429
1022 https://github.com/RPi-Distro/python-gpiozero/issues/465
1023 https://github.com/RPi-Distro/python-gpiozero/issues/468
1024 https://github.com/RPi-Distro/python-gpiozero/issues/575
1025 https://github.com/RPi-Distro/python-gpiozero/issues/459
1026 https://github.com/RPi-Distro/python-gpiozero/issues/468
1027 https://github.com/RPi-Distro/python-gpiozero/issues/484
1028 https://github.com/RPi-Distro/python-gpiozero/issues/469
1029 https://github.com/RPi-Distro/python-gpiozero/issues/523
1030 https://github.com/RPi-Distro/python-gpiozero/issues/520
1031 https://github.com/RPi-Distro/python-gpiozero/issues/434
1032 https://github.com/RPi-Distro/python-gpiozero/issues/565
1033 https://github.com/RPi-Distro/python-gpiozero/issues/576
1034 https://github.com/RPi-Distro/python-gpiozero/issues/558
1035 https://github.com/RPi-Distro/python-gpiozero/issues/497
1036 https://github.com/RPi-Distro/python-gpiozero/issues/504
1037 https://github.com/RPi-Distro/python-gpiozero/issues/529
1038 https://github.com/RPi-Distro/python-gpiozero/issues/504
1039 https://github.com/RPi-Distro/python-gpiozero/issues/529
• Fixed a variety of packaging issues (#535, #518, #519)
• Improved text in factory fallback warnings (#572)

24.5 Release 1.3.2 (2017-03-03)

• Added new Pi models to stop pi_info() (page 189) breaking
• Fix issue with pi_info() (page 189) breaking on unknown Pi models

24.6 Release 1.3.1 (2016-08-31 . . . later)

• Fixed hardware SPI support which Dave broke in 1.3.0. Sorry!
• Some minor docs changes

24.7 Release 1.3.0 (2016-08-31)

• Added ButtonBoard (page 145) for reading multiple buttons in a single class (#340)
• Added Servo (page 123) and AngularServo (page 124) classes for controlling simple servo motors (#248)
• Lots of work on supporting easier use of internal and third-party pin implementations (#359)
• Robot (page 155) now has a proper value (page 156) attribute (#305)
• Added CPUSTemperature (page 169) as another demo of “internal” devices (#294)
• A temporary work-around for an issue with DistanceSensor (page 101) was included but a full fix is in the works (#385)
• More work on the documentation (#320, #295, #289, etc.)

Not quite as much as we’d hoped to get done this time, but we’re rushing to make a Raspbian freeze. As always, thanks to the community - your suggestions and PRs have been brilliant and even if we don’t take stuff exactly as is, it’s always great to see your ideas. Onto 1.4!

24.8 Release 1.2.0 (2016-04-10)

• Added Energenie (page 159) class for controlling Energenie plugs (#69)
• Added LineSensor (page 96) class for single line-sensors (#109)

1038 https://github.com/RPi-Distro/python-gpiozero/issues/535
1039 https://github.com/RPi-Distro/python-gpiozero/issues/518
1040 https://github.com/RPi-Distro/python-gpiozero/issues/519
1041 https://github.com/RPi-Distro/python-gpiozero/issues/572
1042 https://github.com/RPi-Distro/python-gpiozero/issues/340
1043 https://github.com/RPi-Distro/python-gpiozero/issues/248
1044 https://github.com/RPi-Distro/python-gpiozero/issues/359
1045 https://github.com/RPi-Distro/python-gpiozero/issues/305
1046 https://github.com/RPi-Distro/python-gpiozero/issues/294
1047 https://github.com/RPi-Distro/python-gpiozero/issues/385
1048 https://github.com/RPi-Distro/python-gpiozero/issues/320
1049 https://github.com/RPi-Distro/python-gpiozero/issues/295
1050 https://github.com/RPi-Distro/python-gpiozero/issues/289
1051 https://github.com/RPi-Distro/python-gpiozero/issues/89
1052 https://github.com/RPi-Distro/python-gpiozero/issues/109
• Added DistanceSensor (page 101) class for HC-SR04 ultra-sonic sensors (#114)
• Added SnowPi (page 161) class for the Ryanteck Snow-pi board (#130)
• Added when_held (page 95) (and related properties) to Button (page 93) (#115)
• Fixed issues with installing GPIO Zero for python 3 on Raspbian Wheezy releases (#140)
• Added support for lots of ADC chips (MCP3xxx family) (#162) - many thanks to pcopa and lurch!
• Added support for pigpiod as a pin implementation with PiGPIOPin (page 208) (#180)
• Many refinements to the base classes mean more consistency in composite devices and several bugs squashed (#164, #175, #182, #189, #193, #229)
• GPIO Zero is now aware of what sort of Pi it’s running on via pi_info() (page 189) and has a fairly extensive database of Pi information which it uses to determine when users request impossible things (like pull-down on a pin with a physical pull-up resistor) (#222)
• The source/values system was enhanced to ensure normal usage doesn’t stress the CPU and lots of utilities were added (#181, #251)

And I’ll just add a note of thanks to the many people in the community who contributed to this release: we’ve had some great PRs, suggestions, and bug reports in this version. Of particular note:

• Schelto van Doorn was instrumental in adding support for numerous ADC chips
• Alex Eames generously donated a RasPiO Analog board which was extremely useful in developing the software SPI interface (and testing the ADC support)
• Andrew Scheller squashed several dozen bugs (usually a day or so after Dave had introduced them :)

As always, thanks to the whole community - we look forward to hearing from you more in 1.3!

24.9 Release 1.1.0 (2016-02-08)

• Documentation converted to reST and expanded to include generic classes and several more recipes (#80, #82, #101, #119, #135, #168)
• New CamJamKitRobot (page 158) class with the pre-defined motor pins for the new CamJam EduKit
• New LEDBarGraph (page 144) class (many thanks to Martin O’Hanlon!) (#126, #176)

103 https://github.com/RPi-Distro/python-gpiozero/issues/114
104 https://github.com/RPi-Distro/python-gpiozero/issues/130
105 https://github.com/RPi-Distro/python-gpiozero/issues/115
106 https://github.com/RPi-Distro/python-gpiozero/issues/140
107 https://github.com/RPi-Distro/python-gpiozero/issues/162
108 https://github.com/RPi-Distro/python-gpiozero/issues/180
109 https://github.com/RPi-Distro/python-gpiozero/issues/164
110 https://github.com/RPi-Distro/python-gpiozero/issues/175
111 https://github.com/RPi-Distro/python-gpiozero/issues/182
112 https://github.com/RPi-Distro/python-gpiozero/issues/189
113 https://github.com/RPi-Distro/python-gpiozero/issues/193
114 https://github.com/RPi-Distro/python-gpiozero/issues/229
115 https://github.com/RPi-Distro/python-gpiozero/issues/222
116 https://github.com/RPi-Distro/python-gpiozero/issues/181
117 https://github.com/RPi-Distro/python-gpiozero/issues/251
118 https://github.com/RPi-Distro/python-gpiozero/issues/80
119 https://github.com/RPi-Distro/python-gpiozero/issues/82
120 https://github.com/RPi-Distro/python-gpiozero/issues/101
121 https://github.com/RPi-Distro/python-gpiozero/issues/119
122 https://github.com/RPi-Distro/python-gpiozero/issues/135
123 https://github.com/RPi-Distro/python-gpiozero/issues/168
124 https://github.com/RPi-Distro/python-gpiozero/issues/126
125 https://github.com/RPi-Distro/python-gpiozero/issues/176
24.10 Release 1.0.0 (2015-11-16)

- Debian packaging added (#44
- PWMLED (page 113) class added (#58
- TemperatureSensor removed pending further work (#93
- Buzzer.beep() (page 118) alias method added (#75
- Motor (page 120) PWM devices exposed, and Robot (page 155) motor devices exposed (#107

24.11 Release 0.9.0 (2015-10-25)

Fourth public beta

- Added source and values properties to all relevant classes (#76
- Fix names of parameters in Motor (page 120) constructor (#79
- Added wrappers for LED groups on add-on boards (#81

24.12 Release 0.8.0 (2015-10-16)

Third public beta
- Added generic `AnalogInputDevice` (page 139) class along with specific classes for the `MCP3008` (page 135) and `MCP3004` (page 135) (#411093)
- Fixed `DigitalOutputDevice.blink()` (page 127) (#571094)

24.13 Release 0.7.0 (2015-10-09)

Second public beta

24.14 Release 0.6.0 (2015-09-28)

First public beta

24.15 Release 0.5.0 (2015-09-24)

24.16 Release 0.4.0 (2015-09-23)

24.17 Release 0.3.0 (2015-09-22)

24.18 Release 0.2.0 (2015-09-21)

Initial release

1093 https://github.com/RPi-Distro/python-gpiozero/issues/41
1094 https://github.com/RPi-Distro/python-gpiozero/issues/57
CHAPTER 25

License

Copyright © 2015-2019 Ben Nuttall <ben@raspberrypi.org> and contributors; see gpiozero (page 1) for current list

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.
gpiozero, 3  
gpiozero.boards, 141  
gpiozero.devices, 173  
gpiozero.exc, 211  
gpiozero.input_devices, 93  
gpiozero.internal_devices, 167  
gpiozero.output_devices, 111  
gpiozero.pins, 195  
gpiozero.pins.data, 189  
gpiozero.pins.local, 206  
gpiozero.pins.mock, 209  
gpiozero.pins.native, 208  
gpiozero.pins.pi, 205  
gpiozero.pins.pigpio, 207  
gpiozero.pins.rpigpio, 207  
gpiozero.pins.rpio, 207  
gpiozero.spi_devices, 133  
gpiozero.tones, 187  
gpiozero.tools, 179
Symbols
-c, -color
pinout command line option, 71
-h, -help
pinout command line option, 71
-m, -monochrome
pinout command line option, 71
-r REVISION, -revision REVISION
pinout command line option, 71
-x, -xyz
pinout command line option, 71
_shared_key() (gpiozero.SharedMixin class method), 176

A
absoluted() (in module gpiozero.tools), 179
active_high (gpiozero.OutputDevice attribute), 130
active_time (gpiozero.DigitalInputDevice attribute), 105
active_time (gpiozero.EventsMixin attribute), 177
all_values() (in module gpiozero.tools), 182
alternating_values() (in module gpiozero.tools), 184
amber (gpiozero.TrafficLights attribute), 148
AnalogInputDevice (class in gpiozero), 139
angle (gpiozero.AngularServo attribute), 126
AngularServo (class in gpiozero), 124
any_values() (in module gpiozero.tools), 183
arms (gpiozero.SnowPi attribute), 162
averaged() (in module gpiozero.tools), 183

B
backward() (gpiozero.Motor method), 121
backward() (gpiozero.PhaseEnableMotor method), 122
backward() (gpiozero.PhaseEnableRobot method), 157
backward() (gpiozero.Robot method), 155
BadEventHandler, 212
BadPinFactory, 212
BadQueueLen, 212
BadWaitTime, 212
beep() (gpiozero.Buzzer method), 118
bits (gpiozero.AnalogInputDevice attribute), 139
bits_per_word (gpiozero.SPI attribute), 202
blink() (gpiozero.DigitalOutputDevice method), 127
blink() (gpiozero.LED method), 112
blink() (gpiozero.LEDBoard method), 142
blink() (gpiozero.PWMLED method), 113
blink() (gpiozero.PWMOOutputDevice method), 128
blink() (gpiozero.RGBLED method), 116
blue (gpiozero.RGBLED attribute), 117
bluetooth (gpiozero.PiBoardInfo attribute), 191
board (gpiozero.PiBoardInfo attribute), 192
booleanized() (in module gpiozero.tools), 179
bounce (gpiozero.Pin attribute), 200
Button (class in gpiozero), 93
button (gpiozero.StatusBoard attribute), 161
button (gpiozero.TrafficLightsBuzzer attribute), 149
ButtonBoard (class in gpiozero), 145
Buzzer (class in gpiozero), 117
buzzer (gpiozero.JamHat attribute), 154
buzzer (gpiozero.TrafficLightsBuzzer attribute), 149

C
CallbackSetToNone, 216
CamJamKitRobot (class in gpiozero), 158
channel (gpiozero.MCP3002 attribute), 134
channel (gpiozero.MCP3004 attribute), 135
channel (gpiozero.MCP3008 attribute), 135
channel (gpiozero.MCP3024 attribute), 136
channel (gpiozero.MCP3204 attribute), 136
channel (gpiozero.MCP3208 attribute), 136
channel (gpiozero.MCP3304 attribute), 137
channel (gpiozero.MCP3308 attribute), 137
clocked() (in module gpiozero.tools), 180
clock_mode (gpiozero.SPI attribute), 203
clock_phase (gpiozero.SPI attribute), 203
clock_polarity (gpiozero.SPI attribute), 203
close() (gpiozero.CompositeDevice method), 165
close() (gpiozero.Device method), 175
close() (gpiozero.Factory method), 198
close() (gpiozero.GPIODevice method), 108
close() (gpiozero.Pin method), 200
close() (gpiozero.pins.pi.PiFactory method), 205
MockPin (class in gpiozero.pins.mock), 209
MockPWMPin (class in gpiozero.pins.mock), 209
MockTriggerPin (class in gpiozero.pins.mock), 209
model (gpiozero.PiBoardInfo attribute), 191
motion_detected (gpiozero.MotionSensor attribute), 98
MotionSensor (class in gpiozero), 97
Motor (class in gpiozero), 120
multiplied() (in module gpiozero.tools), 183

N
name (gpiozero.HeaderInfo attribute), 192
namedtuple (gpiozero.CompositeDevice attribute), 166
NativeFactory (class in gpiozero.pins.native), 208
NativePin (class in gpiozero.pins.native), 208
negated() (in module gpiozero.tools), 180
nose (gpiozero.SnowPi attribute), 162
note (gpiozero.tones.Tone attribute), 188
number (gpiozero.PinInfo attribute), 193

O
octaves (gpiozero.TonalBuzzer attribute), 120
off () (gpiozero.Buzzer method), 118
off () (gpiozero.CompositeOutputDevice method), 164
off () (gpiozero.DigitalOutputDevice method), 127
off () (gpiozero.Energien method), 159
off () (gpiozero.JamHat method), 154
off () (gpiozero.LED method), 112
off () (gpiozero.LEDBoard method), 143
off () (gpiozero.OutputDevice method), 130
off () (gpiozero.PWMLED method), 114
off () (gpiozero.PWMOutputDevice method), 129
off () (gpiozero.RGBLED method), 116
on () (gpiozero.Buzzer method), 118
on () (gpiozero.CompositeOutputDevice method), 164
on () (gpiozero.DigitalOutputDevice method), 128
on () (gpiozero.Energien method), 159
on () (gpiozero.JamHat method), 154
on () (gpiozero.LED method), 112
on () (gpiozero.LEDBoard method), 143
on () (gpiozero.OutputDevice method), 130
on () (gpiozero.PWMLED method), 114
on () (gpiozero.PWMOutputDevice method), 129
on () (gpiozero.RGBLED method), 116
output_with_state() (gpiozero.Pin method), 200
OutputDevice (class in gpiozero), 130
OutputDeviceBadValue, 214
OutputDeviceError, 214

P
partial (gpiozero.SmoothedInputDevice attribute), 107
pcb_revision (gpiozero.PiBoardInfo attribute), 191
PhaseEnableMotor (class in gpiozero), 122
PhaseEnableRobot (class in gpiozero), 156
physical_pin() (gpiozero.PiBoardInfo method), 190
physical_pins() (gpiozero.PiBoardInfo method), 190
pi_info (gpiozero.Factory attribute), 199
pi_info() (in module gpiozero), 189
PiBoardInfo (class in gpiozero), 189
PiFactory (class in gpiozero.pins.pi), 205
PIGPIO_ADDR, 46, 56, 197
PiGPIOFactory (class in gpiozero.pins.pi), 164
PiGPIOPin (class in gpiozero.pins.pi), 208
PiHutXmasTree (class in gpiozero), 149
PiColorConverters (class in gpiozero), 150
PiColorConverters (class in gpiozero), 150
PiColorConverters (class in gpiozero), 150
PiColorConverters (class in gpiozero), 150
PiColorConverters (class in gpiozero), 150
PiColorConverters (class in gpiozero), 150
Pin (class in gpiozero), 199
Pin (gpiozero.Button attribute), 95
Pin (gpiozero.Buzzer attribute), 119
Pin (gpiozero.GPJODevice attribute), 109
Pin (gpiozero.LED attribute), 112
Pin (gpiozero.LineSensor attribute), 100
Pin (gpiozero.LineSensor attribute), 97
Pin (gpiozero.MotionSensor attribute), 99
Pin (gpiozero.PWMLED attribute), 114
pin() (gpiozero.Factory method), 198
pin() (gpiozero.pins.pin.Factory method), 209
pin() (gpiozero.pins.pi.PinFactory method), 205
pin_class (gpiozero.pins.mock.MockFactory attribute), 209
pin_factory (gpiozero.Device attribute), 175
PinEdgeDetectUnsupported, 214
PinError, 214
PinFactoryFallback, 216
PinFixedPull, 214
PingServer (class in gpiozero), 168
PinInfo (class in gpiozero), 193
PinInvalidBounce, 214
PinInvalidEdges, 214
PinInvalidFunction, 214
PinInvalidPin, 215
PinInvalidPull, 214
PinInvalidState, 214
PinMultiplePins, 215
PinNonPhysical, 216
PinNoPins, 215
pinout command line option
  -c, -color, 71
  -h, -help, 71
  -m, -monochrome, 71
  -r REVISION, -revision REVISION, 71
  -x, -xyz, 71
PinPWMError, 215
PinPMMFixedValue, 215
PinPWMUnsupported, 215
pins (gpiozero.HeaderInfo attribute), 192
PinSetInput, 214
Gpiozero Documentation, Release 1.5.1

start_time (giozero.TimeOfDay attribute), 168
state (giozero.Pin attribute), 201
StatusBoard (class in giozero), 160
StatusZero (class in giozero), 160
stop () (giozero.Motor method), 121
stop () (giozero.PhaseEnableMotor method), 122
stop () (giozero.PhaseEnableRobot method), 157
stop () (giozero.Robot method), 156
stop () (giozero.TonalBuzzer method), 120
storage (giozero.PiBoardInfo attribute), 191
summed () (in module giozero.tools), 184

T
temperature (giozero.CPUTemperature attribute), 169
threshold (giozero.SmoothedInputDevice attribute), 107
threshold_distance (giozero.DistanceSensor attribute), 103
ThresholdOutOfRange, 216
ticks () (giozero.Factory method), 199
ticks () (giozero.pins.local.LocalPiFactory static method), 206
ticks_diff () (giozero.Factory method), 199
ticks_diff () (giozero.pins.local.LocalPiFactory static method), 206
TimeOfDay (class in giozero), 167
to_gpio () (giozero.PiBoardInfo method), 190
toggle () (giozero.Buzzer method), 119
toggle () (giozero.CompositeOutputDevice method), 164
toggle () (giozero.LED method), 112
toggle () (giozero.LEDBoard method), 143
toggle () (giozero.OutputDevice method), 130
toggle () (giozero.PWMLED method), 114
toggle () (giozero.PWMOutputDevice method), 129
toggle () (giozero.RGBLED method), 117
TonalBuzzer (class in giozero), 119
Tone (class in giozero.tones), 187
tone (giozero.TonalBuzzer attribute), 120
TrafficHat (class in giozero), 153
TrafficLights (class in giozero), 147
TrafficLightsBuzzer (class in giozero), 148
transfer () (giozero.SPI method), 202
trigger (giozero.DistanceSensor attribute), 103

U
up () (giozero.tones.Tone method), 188
usage (giozero.DiskUsage attribute), 171
usb (giozero.PiBoardInfo attribute), 191
utc (giozero.TimeOfDay attribute), 168

V
value (giozero.AnalogInputDevice attribute), 139
value (giozero.AnalogInputDevice attribute), 139
value (giozero.AngularServo attribute), 126
value (giozero.Button attribute), 95
value (giozero.ButtonBoard attribute), 147
value (giozero.Buzzer attribute), 119
value (giozero.CompoudDevice attribute), 166
value (giozero.CompositeOutputDevice attribute), 164
value (giozero.CPUTemperature attribute), 169
value (giozero.Device attribute), 175
value (giozero.DigitalInputDevice attribute), 105
value (giozero.DigitalOutputDevice attribute), 128
value (giozero.DiskUsage attribute), 171
value (giozero.DistanceSensor attribute), 103
value (giozero.Energenie attribute), 159
value (giozero.GPIODevice attribute), 109
value (giozero.InputDevice attribute), 108
value (giozero.Led attribute), 112
value (giozero.LEDBarGraph attribute), 145
value (giozero.LightSensor attribute), 100
value (giozero.LineSensor attribute), 97
value (giozero.LoadAverage attribute), 170
value (giozero.MCP3001 attribute), 134
value (giozero.MCP3002 attribute), 134
value (giozero.MCP3004 attribute), 135
value (giozero.MCP3008 attribute), 135
value (giozero.MCP3201 attribute), 135
value (giozero.MCP3202 attribute), 136
value (giozero.MCP3204 attribute), 136
value (giozero.MCP3208 attribute), 137
value (giozero.MCP3301 attribute), 137
value (giozero.MCP3302 attribute), 137
value (giozero.MCP3304 attribute), 138
value (giozero.MotionSensor attribute), 99
value (giozero.Motor attribute), 121
value (giozero.OutputDevice attribute), 130
value (giozero.PhaseEnableMotor attribute), 123
value (giozero.PhaseEnableRobot attribute), 157
value (giozero.PingServer attribute), 169
value (giozero.PWMLED attribute), 114
value (giozero.PWMOutputDevice attribute), 129
value (giozero.RGBLED attribute), 117
value (giozero.Robot attribute), 156
value (giozero.Servo attribute), 124
value (giozero.SmoothedInputDevice attribute), 107
value (giozero.TimeOfDay attribute), 168
value (giozero.TonalBuzzer attribute), 120
values (giozero.LEDBarGraph attribute), 145
values (giozero.ValuesMixin attribute), 176
ValuesMixin (class in giozero), 176
voltage (giozero.AnalogInputDevice attribute), 139

W
wait_for_active () (giozero.DigitalInputDevice method), 105
wait_for_active () (giozero.EventsMixin method), 177
wait_for_dark () (giozero.LightSensor method), 100
wait_for_in_range () (giozero.DistanceSensor method), 102
wait_for_inactive() (gpiozero.DigitalInputDevice method), 105
wait_for_inactive() (gpiozero.EventsMixin method), 177
wait_for_light() (gpiozero.LightSensor method), 100
wait_for_line() (gpiozero.LineSensor method), 96
wait_for_motion() (gpiozero.MotionSensor method), 98
wait_for_no_line() (gpiozero.LineSensor method), 97
wait_for_no_motion() (gpiozero.MotionSensor method), 98
wait_for_out_of_range() (gpiozero.DistanceSensor method), 102
wait_for_press() (gpiozero.Button method), 94
wait_for_press() (gpiozero.ButtonBoard method), 146
wait_for_release() (gpiozero.Button method), 94
wait_for_release() (gpiozero.ButtonBoard method), 146
when_activated (gpiozero.DigitalInputDevice attribute), 105
when_activated (gpiozero.EventsMixin attribute), 177
when_changed (gpiozero.Pin attribute), 201
when_dark (gpiozero.LightSensor attribute), 100
when_deactivated (gpiozero.DigitalInputDevice attribute), 105
when_deactivated (gpiozero.EventsMixin attribute), 177
when_held (gpiozero.Button attribute), 95
when_held (gpiozero.HoldMixin attribute), 178
when_in_range (gpiozero.HoldMixin attribute), 178
when_light (gpiozero.LightSensor attribute), 101
when_line (gpiozero.LineSensor attribute), 97
when_motion (gpiozero.MotionSensor attribute), 99
when_no_line (gpiozero.LineSensor attribute), 97
when_no_motion (gpiozero.MotionSensor attribute), 99
when_out_of_range (gpiozero.DistanceSensor attribute), 103
when_pressed (gpiozero.Button attribute), 95
when_pressed (gpiozero.ButtonBoard attribute), 147
when_released (gpiozero.Button attribute), 95
when_released (gpiozero.ButtonBoard attribute), 147
wifi (gpiozero.PiBoardInfo attribute), 191
write() (gpiozero.SPI method), 202

Y
yellow (gpiozero.TrafficLights attribute), 148