gpiozero 1.6.1 Documentation

Ben Nuttall

Nov 19, 2021
## Contents

1  Installing GPIO Zero  
2  Basic Recipes  
3  Advanced Recipes  
4  Configuring Remote GPIO  
5  Remote GPIO Recipes  
6  Pi Zero USB OTG  
7  Source/Values  
8  Command-line Tools  
9  Frequently Asked Questions  
10 Backwards Compatibility  
11 Migrating from RPi.GPIO  
12 Contributing  
13 Development  
14 API - Input Devices  
15 API - Output Devices  
16 API - SPI Devices  
17 API - Boards and Accessories  
18 API - Internal Devices  
19 API - Generic Classes  
20 API - Device Source Tools  
21 API - Fonts  
22 API - Tones
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>23 API - Pi Information</td>
<td>221</td>
</tr>
<tr>
<td>24 API - Pins</td>
<td>227</td>
</tr>
<tr>
<td>25 API - Exceptions</td>
<td>247</td>
</tr>
<tr>
<td>26 Changelog</td>
<td>253</td>
</tr>
<tr>
<td>27 License</td>
<td>263</td>
</tr>
<tr>
<td>Python Module Index</td>
<td>265</td>
</tr>
<tr>
<td>Index</td>
<td>267</td>
</tr>
</tbody>
</table>
Installing GPIO Zero

GPIO Zero is installed by default in the Raspberry Pi OS\(^1\) desktop image, Raspberry Pi OS\(^2\) Lite image, and the Raspberry Pi Desktop\(^3\) image for PC/Mac, all available from raspberrypi.org\(^4\). Follow these guides to installing on other operating systems, including for PCs using the remote GPIO (page 49) feature.

1.1 Raspberry Pi

GPIO Zero is packaged in the apt repositories of Raspberry Pi OS, Debian\(^5\) and Ubuntu\(^6\). It is also available on PyPI\(^7\).

1.1.1 apt

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
```

---

\(^1\) [https://www.raspberrypi.org/software/operating-systems/](https://www.raspberrypi.org/software/operating-systems/)


\(^3\) [https://www.raspberrypi.org/software/raspberry-pi-desktop/](https://www.raspberrypi.org/software/raspberry-pi-desktop/)


\(^5\) [https://packages.debian.org/buster/python3-gpiozero](https://packages.debian.org/buster/python3-gpiozero)

\(^6\) [https://packages.ubuntu.com/hirsute/python3-gpiozero](https://packages.ubuntu.com/hirsute/python3-gpiozero)

\(^7\) [https://pypi.org/project/gpiozero/](https://pypi.org/project/gpiozero/)
1.1.2 pip

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` 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 103) 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 49) page for more information.

1.3 Documentation

This documentation is also available for offline installation like so:

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

This will install the HTML version of the documentation under the `/usr/share/doc/python-gpiozero-doc/html` path. To view the offline documentation you have several options:

You can open the documentation directly by visiting `file:///usr/share/doc/python-gpiozero-doc/html/index.html` in your browser. However, be aware that using `file://` URLs sometimes breaks certain elements. To avoid this, you can view the docs from an `http://` style URL by starting a trivial HTTP server with Python, like so:

```
$ python3 -m http.server -d /usr/share/doc/python-gpiozero-doc/html
```

Then visit `http://localhost:8000/` in your browser.

Alternatively, the package also integrates into Debian’s doc-base system, so you can install one of the doc-base clients (dochelp, dwww, dhelp, doc-central, etc.) and use its interface to locate this document.

If you want to view the documentation offline on a different device, such as an eReader, there are Epub and PDF versions of the documentation available for download from the ReadTheDocs site. Simply click on the “Read the Docs” box at the bottom-left corner of the page (under the table of contents) and select “PDF” or “Epub” from the “Downloads” section.

---

9 https://wiki.debian.org/doc-base
10 https://gpiozero.readthedocs.io/
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:

11 [https://pypi.python.org/pypi/RPi.GPIO](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\(^{12}\) 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)
>>> led = LED(“GPIO17”)  
>>> led = LED(“BCM17”)  
>>> led = LED(“BOARD11”)  
```

\(^{12}\)https://projects.drogon.net/raspberry-pi/wiringpi/pins/

(continues on next page)
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("WPI0")
>>> led = LED("J8:11")
```

```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 127) 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()`\textsuperscript{13}. See *How do I keep my script running?* (page 81) for more information.

## 2.4 LED with variable brightness

Any regular LED can have its brightness value set using PWM (pulse-width-modulation). In GPIO Zero, this can be achieved using `PWMLED` (page 129) 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)
```

\textsuperscript{13} [https://docs.python.org/3.7/library/signal.html#signal.pause](https://docs.python.org/3.7/library/signal.html#signal.pause)
Similarly to blinking on and off continuously, a PWMLED can pulse (fade in and out continuously):

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

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

## 2.5 Button

Check if a Button (page 107) is pressed:

```python
from gpiozero import Button

button = Button(2)

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

Wait for a button to be pressed before continuing:

```python
from gpiozero import Button

button = Button(2)

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

Run a function every time the button is pressed:
```python
from gpiozero import Button
from signal import pause

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

button = Button(2)

button.when_pressed = say_hello

pause()
```

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

Similarly, functions can be attached to button releases:

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

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

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

button = Button(2)

button.when_pressed = say_hello
button.when_released = say_goodbye

pause()
```

14 [https://docs.python.org/3.7/library/constants.html#None](https://docs.python.org/3.7/library/constants.html#None)
2.6 Button controlled LED

Turn on an LED (page 127) when a Button (page 107) 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 to take a picture using `button.when_pressed = camera.capture()` would not work because the `capture()` method requires an output parameter. However, this can be achieved using a custom function which requires no parameters:
```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(f'/home/pi/{timestamp}.jpg'.format(timestamp=timestamp))

button.when_pressed = capture

pause()
```

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(f'/home/pi/{timestamp}.jpg'.format(timestamp=timestamp))

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

pause()
```

### 2.8 Shutdown button

The `Button` (page 107) 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

pause()
```
A collection of LEDs can be accessed using `LEDBoard` (page 159):

```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 159) 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 159) examples in the *advanced LEDBoard recipes* (page 39).
2.10 LEDBarGraph

A collection of LEDs can be treated like a bar graph using `LEDBarGraph` (page 162):

```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 162) 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)
```

(continues on next page)
from gpiozero import LEDCharDisplay
from time import sleep

display = LEDCharDisplay(21, 20, 16, 22, 23, 24, 12, dp=25)

for char in '321GO':
    display.value = char
    sleep(1)

display.off()

Alternatively:

from gpiozero import LEDCharDisplay
from signal import pause

display = LEDCharDisplay(21, 20, 16, 22, 23, 24, 12, dp=25)
display.source_delay = 1
display.source = '321G0'

A common 7-segment display can be used to represent a variety of characters using LEDCharDisplay (page 164) (which actually supports an arbitrary number of segments):

https://en.wikipedia.org/wiki/Seven-segment_display
See a multi-character example in the *advanced recipes* (page 41) chapter.

## 2.12 Traffic Lights

A full traffic lights system.

Using a *TrafficLights* (page 170) 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()
```

(continues on next page)
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 127) 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()
```
2.13 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()

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

See Push Button Stop Motion\(^{16}\) for a full resource.

2.14 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)
```

```python
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\(^{17}\) for a full resource.

### 2.15 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\(^\text{18}\) for a full resource.

## 2.16 All on when pressed

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

**FishDish** (page 175):

```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 176):

```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 127), **Buzzer** (page 134), and **Button** (page 107) 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()

## 2.17 Full color LED

Making colours with an `RGBLED` (page 131):

```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
```

(continues on next page)
Light an LED (page 127) when a MotionSensor (page 112) 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()

2.19 Light sensor

Have a LightSensor (page 113) detect light and dark:

```
from gpiozero import LightSensor

sensor = LightSensor(18)

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

Run a function when the light changes:

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

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

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

Or make a *PWMLED* (page 129) 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()
```

## 2.20 Distance sensor

Have a *DistanceSensor* (page 115) detect the distance to the nearest object:

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

sensor = DistanceSensor(23, 24)

while True:
    print(sensor.distance)
    sleep(0.1)
```

**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!)
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()
```

### 2.21 Rotary encoder

![Rotary encoder diagram]

**Note:** In this recipe, I’ve used a common *anode* RGB LED. Often, Pi projects use common *cathode* RGB LEDs because they’re slightly easier to think about electrically. However, in this case all three components can be found in an illuminated rotary encoder which incorporates a common anode RGB LED, and a momentary push button. This is also the reason for the button being wired active-low, contrary to most other examples in this documentation.

For the sake of clarity, the diagram shows the three separate components, but this same circuit will work equally well with this commonly available illuminated rotary encoder\(^\text{19}\) instead.

Have a rotary encoder, an RGB LED, and button act as a color picker.

from threading import Event
from colorzero import Color
from gpiozero import RotaryEncoder, RGBLED, Button

rotor = RotaryEncoder(16, 20, wrap=True, max_steps=180)
rotor.steps = -180
led = RGBLED(22, 23, 24, active_high=False)
btn = Button(21, pull_up=False)
led.color = Color('#f00')
done = Event()

def change_hue():
    # Scale the rotor steps (-180..180) to 0..1
    hue = (rotor.steps + 180) / 360
    led.color = Color(h=hue, s=1, v=1)

def show_color():
    print('Hue {led.color.hue.deg:.1f}° = {led.color.html}'.format(led=led))

def stop_script():
    print('Exiting')
done.set()

print('Select a color by turning the knob')
rotor.when_rotated = change_hue
print('Push the button to see the HTML code for the color')
btn.when_released = show_color
print('Hold the button to exit')
btn.when_held = stop_script
done.wait()

2.22 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:

```python
from gpiozero import Servo
from gpiozero.tools import sin_values
from signal import pause

servo = Servo(17)

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

pause()
```

Use `AngularServo` (page 141) so you can specify an angle:

```python
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)
```
Spin a Motor (page 137) 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)
```
2.24 Robot

Make a Robot (page 179) drive around in (roughly) a square:

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

robot = Robot(left=Motor(4, 14), right=Motor(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, Motor, DistanceSensor
from signal import pause

... (continues on next page)
sensor = DistanceSensor(23, 24, max_distance=1, threshold_distance=0.2)
robot = Robot(left=Motor(4, 14), right=Motor(17, 18))
sensor.when_in_range = robot.backward
sensor.when_out_of_range = robot.stop
pause()

2.25 Button controlled robot

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

from gpiozero import Robot, Motor, Button
from signal import pause
robot = Robot(left=Motor(4, 14), right=Motor(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()
2.26 Keyboard controlled robot

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

```python
import curses
from gpiozero import Robot, Motor

robot = Robot(left=Motor(4, 14), right=Motor(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
```

(continues on next page)
```
while True:
    curses.halfdelay(1)
    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:

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

robot = Robot(left=Motor(4, 14), right=Motor(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,
}
```
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.
2.27 Motion sensor robot

Make a robot drive forward when it detects motion:

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

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

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

pause()
```

Alternatively:
from gpiozero import Robot, Motor, MotionSensor
from gpiozero.tools import zip_values
from signal import pause

robot = Robot(left=Motor(4, 14), right=Motor(17, 18))
pir = MotionSensor(5)
robot.source = zip_values(pir, pir)
pause()

2.28 Potentiometer

Continually print the value of a potentiometer (values between 0 and 1) connected to a MCP3008 (page 151) analog to digital converter:

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

Present the value of a potentiometer on an LED bar graph using PWM to represent states that won’t “fill” an LED:
from gpiozero import LEDBarGraph, MCP3008
from signal import pause

graph = LEDBarGraph(5, 6, 13, 19, 26, pwm=True)
pot = MCP3008(channel=0)

graph.source = pot

pause()

## 2.29 Measure temperature with an ADC

Wire a TMP36 temperature sensor to the first channel of an *MCP3008* (page 151) 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)
```
2.30 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 204) 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)
```

---

21 [https://docs.python.org/3.7/reference/compound_stmts.html#while](https://docs.python.org/3.7/reference/compound_stmts.html#while)
blue_pot = MCP3008(2)
led.source = zip_values(red_pot, green_pot, blue_pot)
pause()

2.31 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\(^\text{22}\) to remotely control the lamp, and the TimeOfDay (page 192) 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))
daytime.when_activated = lamp.on
daytime.when_deactivated = lamp.off

pause()
```

2.32 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 193) 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')
google.when_activated = green.on
google.when_deactivated = green.off
red.source = negated(green)
pause()
```

2.33 CPU Temperature Bar Graph

You can read the Raspberry Pi's own CPU temperature using the built-in CPUtemperature (page 194) class, and display this on a “bar graph” of LEDs:

\(^{22}\)https://energenie4u.co.uk/catalogue/product/ENER002-2PI
from gpiozero import LEDBarGraph, CPUPerformance
from signal import pause

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

leds.source = cpu

pause()
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 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 also supports indexing. This means you can access the individual LED 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:
from gpiozero import LEDBoard
from time import sleep

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

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

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

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

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

LEDBoard (page 159) objects can have their LED objects named upon construction. This means the individual LEDs can be accessed by their name:

from gpiozero import LEDBoard
from time import sleep

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

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

LEDBoard (page 159) objects can also be nested within other LEDBoard (page 159) objects:

from gpiozero import LEDBoard
from time import sleep

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

leds.red.on()  # both reds on
sleep(1)
leds.green.on()  # both greens on
sleep(1)
leds.off()  # all off
sleep(1)
leds.red.top.on()  # top red on
sleep(1)
leds.green.bottom.on()  # bottom green on
sleep(1)
### 3.2 Multi-character 7-segment display

The 7-segment display demonstrated in the previous chapter is often available in multi-character variants (typically 4 characters long). Such displays are multiplexed meaning that the LED pins are typically the same as for the single character display but are shared across all characters. Each character in turn then has its own common line which can be tied to ground (in the case of a common cathode display) to enable that particular character. By activating each character in turn very quickly, the eye can be fooled into thinking four different characters are being displayed simultaneously.

In such circuits you should not attempt to sink all the current from a single character (which may have up to 8 LEDs, in the case of a decimal-point, active) into a single GPIO. Rather, use some appropriate transistor (or similar component, e.g. an opto-coupler) to tie the digit’s cathode to ground, and control that component from a GPIO.

This circuit demonstrates a 4-character 7-segment (actually 8-segment, with decimal-point) display, controlled by the Pi’s GPIOs with 4 2N-3904 NPN transistors to control the digits.

**Warning:** You are strongly advised to check the data-sheet for your particular multi-character 7-segment display. The pin-outs of these displays vary significantly and are very likely to be different to that shown on the breadboard above. For this reason, the schematic for this circuit is provided below; adapt it to your particular display.
The following code can be used to scroll a message across the display:

```python
from itertools import cycle
from collections import deque
from gpiozero import LEDMultiCharDisplay
from signal import pause

display = LEDMultiCharDisplay(
    LEDCharDisplay(22, 23, 24, 25, 21, 20, 16, dp=12), 26, 19, 13, 6)

def scroller(message, chars=4):
    d = deque(maxlen=chars)
    for c in cycle(message):
        d.append(c)
        if len(d) == chars:
            yield ''.join(d)

display.source_delay = 0.2
display.source = scroller('GPIO ZERO')
pause()
```

### 3.3 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),
)
```

(continues on next page)
```python
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 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()
```

### 3.4 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
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'
```

---

23 [https://thepihut.com/status](https://thepihut.com/status)
red = LED(12)
green = LED(16)

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

pause()

Note this recipe requires travispy\textsuperscript{24}. Install with \texttt{sudo pip3 install travispy}.

### 3.5 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, Motor
from time import sleep
from signal import pause

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

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

instructions = []

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

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

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

pause()
```

\textsuperscript{24} \url{https://travispy.readthedocs.io/}
3.6 Robot controlled by 2 potentiometers

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

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

robot = Robot(left=Motor(4, 14), right=Motor(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, Motor, MCP3008
from gpiozero.tools import scaled
from signal import pause

robot = Robot(left=Motor(4, 14), right=Motor(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.

3.7 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()
    led.on()
    bd.wait_for_release()
    led.off()
```

---

25 [https://docs.python.org/3.7/library/functions.html#zip](https://docs.python.org/3.7/library/functions.html#zip)
Note this recipe requires `bluedot` and the associated Android app. See the BlueDot documentation\textsuperscript{26} for installation instructions.

### 3.8 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, Motor
from signal import pause

bd = BlueDot()
robot = Robot(left=Motor(4, 14), right=Motor(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, Motor
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 min(1, min(-1, v))

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

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

\textsuperscript{26} https://bluedot.readthedocs.io/en/latest/index.html
3.9 Controlling the Pi’s own LEDs

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

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

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

Now you can control the LEDs with gpiozero like so:

```
from gpiozero import LED
from signal import pause

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

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

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

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

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

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

On the Pi 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 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 227) documentation page.

One of the pin libraries supported, pigpio, 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 57) page for examples on how remote pins can be used.

4.1 Preparing the Raspberry Pi

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

$ sudo apt install pigpio

Alternatively, pigpio is available from abyz.me.uk.

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

4.1.1 Enable remote connections

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

---

27 https://pypi.python.org/pypi/RPi.GPIO
28 http://abyz.me.uk/rpi/pigpio/python.html
29 http://abyz.me.uk/rpi/pigpio/download.html
Alternatively, enter `sudo raspi-config` on the command line, and enable Remote GPIO. This is functionally equivalent to the desktop method.

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

### 4.1.2 Command-line: systemctl

To automate running the daemon at boot time, run:

```
$ sudo systemctl enable pigpiod
```

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

```
$ sudo systemctl start pigpiod
```

### 4.1.3 Command-line: pigpiod

Another option is to launch the pigpio daemon manually:

```
$ sudo pigpiod
```

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

```
$ sudo pigpiod -n localhost  # allow localhost only
$ sudo pigpiod -n 192.168.1.65 # allow 192.168.1.65 only
$ sudo pigpiod -n localhost -n 192.168.1.65 # allow localhost and 192.168.1.65 only
```
Note: Note that running `sudo pigpiod` will not honour the Remote GPIO configuration setting (i.e. without the --n flag it will allow remote connections even if the remote setting is disabled), but `sudo systemctl enable pigpiod` or `sudo systemctl start pigpiod` will not allow remote connections unless configured accordingly.

4.2 Preparing the control computer

If the control computer (the computer you’re running your Python code from) is a Raspberry Pi running Raspberry Pi OS (or a PC running Raspberry Pi Desktop x86[30]), then you have everything you need. If you’re using another Linux distribution, Mac OS or Windows then you’ll need to install the pigpio[31] 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-pip[32].)

Next, install GPIO Zero and pigpio for Python 3:

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

[31] http://abyz.me.uk/rpi/pigpio/python.html  
[32] https://pypi.org/project/pip/  
4.2. Preparing the control computer 51)
**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`.

Next, install GPIO Zero and pigpio with pip:

```bash
$ pip3 install gpiozero pigpio
```

Or for Python 2:

```bash
$ 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 guide. Next, install GPIO Zero and pigpio with pip:

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

**4.3 Environment variables**

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

```bash
$ PIGPIO_ADDR=192.168.1.3 python3 hello.py
```

If you are running this from a PC (not a Raspberry Pi) with gpiozero and the pigpio Python library installed, this will work with no further configuration. However, if you are running this from a Raspberry Pi, you will also need to ensure the default pin factory is set to `PiGPIOFactory` (page 242). If `RPi.GPIO` is installed, this will be selected as the default pin factory, so either uninstall it, or use the `GPIOZERO_PIN_FACTORY` environment variable to override it:

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

This usage will set the pin factory to `PiGPIOFactory` (page 242) 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:

35 http://abyz.me.uk/rpi/pigpio/python.html
36 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 242) can be used, as pigpio is the only pin library which supports remote GPIO.

### 4.4 Pin factories

An alternative (or additional) method of configuring gpiozero objects to use remote pins is to create instances of `PiGPIOFactory` (page 242) 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')
```

37 http://abyz.me.uk/rpi/pigpio/python.html
led_1 = LED(17, pin_factory=factory3)
led_2 = LED(17, pin_factory=factory4)

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

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

```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

(continues on next page)
factory3 = PiGPIOFactory(host='192.168.1.3')
factory4 = PiGPIOFactory(host='192.168.1.4')

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

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

Note that these examples use the LED (page 127) 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:

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

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

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

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

remote_factory = PiGPIOFactory(host='192.168.1.3')

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

You could even use a HAT which is not supported by GPIO Zero (such as the Sense HAT38) on one Pi, and use remote pins to control another over the network:

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

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

---

38 https://www.raspberry.org/products/sense-hat/
```python
while True:
    pir.wait_for_motion()
    sense.show_message(sense.temperature)
```

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

### 4.5 Remote GPIO usage

Continue to:

- *Remote GPIO Recipes* (page 57)
- *Pi Zero USB OTG* (page 61)
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 49).

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 107) on one Raspberry Pi control the LED (page 127) of another:

```python
from gpiozero import Button, 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 127) will come on when both buttons are pressed:

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

(factory, button1, button2)
```

(continues on next page)
factory3 = PiGPIOFactory(host='192.168.1.3')
factory4 = PiGPIOFactory(host='192.168.1.4')

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

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

5.3 Multi-room motion alert

Install a Raspberry Pi with a MotionSensor (page 112) in each room of your house, and have an LED indicator showing when there’s motion in each room:

from gpiozero import LEDBoard, MotionSensor
from gpiozero.pins.pigpio import PiGPIOFactory
from 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 134) attached in each room you want to hear the doorbell, and use a push Button (page 107) as the doorbell:

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

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

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

for buzzer in buzzers:
    buzzer.source = button

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 179), 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, Motor
from gpiozero.pins.pigpio import PiGPIOFactory
from signal import pause

factory = PiGPIOFactory(host='192.168.1.17')
robot = Robot(left=Motor(4, 14), right=Motor(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\(^{39}\) (not supported by GPIO Zero) includes temperature, humidity and pressure sensors, but no light sensor. Remote GPIO allows an external LightSensor (page 113) 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:
        # code to change LED

```

(continued from previous page)

```python
    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 and Pi Zero W 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 49) 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 Raspberry Pi OS 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 from raspberrypi.org
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 setup (using Raspberry Pi OS)

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

```bash
$ sudo apt update
$ sudo apt install usbbootgui
```
6.1.3 Ubuntu setup

1. Add the Raspberry Pi PPA to your system:

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

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

```bash
sudo pip3 uninstall gpiozero pigpio
```

3. Install the required packages from the PPA:

```bash
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:

![Raspberry Pi connected](image)

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

Raspberry Pi OS will name your Pi Zero connection `usb0`. On Ubuntu, this will likely be something else. You can ping it using the address `fe80::1%usb0` followed by the connection string. You can look this up using `ifconfig`.

Set the `GPIOZERO_PIN_FACTORY` (page 80) and `PIGPIO_ADDR` (page 80) environment variables on your PC so GPIO Zero connects to the “remote” Pi Zero:

```bash
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 49).
Read more on the GPIO expander in blog posts on raspberrypi.org\textsuperscript{43} and benmuttall.com\textsuperscript{44}.

### 6.2 Legacy method - SD card required

The legacy method requires the Pi Zero to have an SD card with Raspberry Pi OS inserted.
Start by creating a Raspberry Pi OS (desktop or lite) SD card, and then configure the boot partition like so:

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

(See guides on blog.gbaman.info\textsuperscript{45} and learn.adafruit.com\textsuperscript{46} 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 \texttt{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:

\textsuperscript{43} https://www.raspberrypi.org/blog/gpio-expander/
\textsuperscript{44} http://bennuttall.com/raspberry-pi-zero-gpio-expander/
\textsuperscript{45} http://blog.gbaman.info/?p=791
\textsuperscript{46} https://learn.adafruit.com/turning-your-raspberry-pi-zero-into-a-usb-gadget/ethernet-gadget
```bash
$ GPIOZERO_PIN_FACTORY=pigpio PIGPIO_ADDR=raspberrypi.local python3 led.py
```
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 203) 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:
Every device also has a `values` (page 204) property (a generator\(^{47}\) continuously yielding the device’s current value). All output devices have a `source` (page 204) property which can be set to any iterator\(^{48}\). 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 204) 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 204) property is used to feed values into the output device. Prior to v1.5, the `source` (page 204) had to be set directly to a device’s `values` (page 204) 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 204) to a device object.

It is also possible to set an output device’s `source` (page 204) 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 potentiometer:

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

led = PWMLED(17)
pot = MCP3008()

led.source = pot

green_led = PWMLED(27)
green_led.source = led

pause()
```
the red LED, so both LEDs will be on when the button is pressed:

![Diagram of LED and Button circuit]

```python
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 204):

![Diagram of Output device, custom generator, and Input device]

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

```python
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:

![Diagram of Output device, custom generator, and Input device]

For example, writing a generator function to randomly yield 0 or 1:
```python
from gpiozero import LED
from random import randint
from signal import pause

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

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

If the iterator is infinite (i.e. an infinite generator), the elements will be processed until the `source` (page 204) 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_delay = 1
led.source = [1, 0, 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 207).

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
```

---

49 https://docs.python.org/3.7/library/constants.html#None
led = PWMLED(4)
led.source = random_values()
led.source_delay = 0.1
pause()

Note that in the above example, `source_delay` (page 204) is used to make the LED iterate over the random values slightly slower. `source_delay` (page 204) 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:

![Diagram showing a single source tool connected to an output device and an input device.]

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 204) derivatives, or iterators, including a device’s `values` (page 204) property.

Some tools combine the values of multiple sources:

![Diagram showing two input devices connected to a source tool.]

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

```
```

---

7.2. Source Tools

---

---
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()

Similarly, any_values() (page 211) with two buttons would simulate an OR\(^{51}\) gate.

While most devices have a value (page 203) range between 0 and 1, some have a range between -1 and 1 (e.g., Motor (page 137), Servo (page 139) and TonalBuzzer (page 135)). Some source tools output values between -1 and 1, which are ideal for these devices, for example passing sin_values() (page 214) in:

```python
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\(^{52}\). 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 204) will alter the speed at which the device iterates through the

\(^{51}\) https://en.wikipedia.org/wiki/OR_gate
\(^{52}\) https://en.wikipedia.org/wiki/Sine_wave
values. Alternatively, the tool `cos_values()` (page 213) 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 191) 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 204) property which can be used to feed values into a device’s `source` (page 204). For example, a lamp connected to an *Energenie* (page 182) socket can be controlled by a *TimeOfDay* (page 192) object so that it is on between the hours of 8am and 8pm:

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

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

daytime.when_activated = lamp.on

daytime.when_deactivated = lamp.off

pause()
```

Using the *DiskUsage* (page 197) class with *LEDBarGraph* (page 162) 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 207) source tool to convert the light sensor from a float value into a boolean:
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 203) of a composite device made up of the nested values of its devices. For example, the value of a Robot (page 179) 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)
>>> robot.forward()
>>> tuple(robot.value)
(1.0, 1.0)
>>> robot.backward()
>>> tuple(robot.value)
(-1.0, -1.0)
>>> robot.value = (1, 1)  # robot is now driven forwards
```

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

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

(continues on next page)
robot = Robot(left=Motor(4, 14), right=Motor(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, Motor, MCP3008
from gpiozero.tools import scaled
from signal import pause

robot = Robot(left=Motor(4, 14), right=Motor(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()`\(^{53}\) rather than the tool `zip_values()` (page 212) as the `scaled()` (page 210) tool yields values which do not need converting, just zipping. Also note that this use of `zip()`\(^{54}\) will not work in Python 2, instead use `izip`\(^{55}\).

---

\(^{53}\) [https://docs.python.org/3.7/library/functions.html#zip](https://docs.python.org/3.7/library/functions.html#zip)

\(^{54}\) [https://docs.python.org/3.7/library/functions.html#zip](https://docs.python.org/3.7/library/functions.html#zip)

\(^{55}\) [https://docs.python.org/2/library/itertools.html#itertools.izip](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.

**Note:** Note that only the Python 3 version of the Debian package includes the pinout command line tool, so as not to create a conflict if both versions are installed. If you only installed the `python-gpiozero` apt package, the pinout tool will not be available. Instead, you can additionally install the `python3-gpiozero` package, or alternatively install gpiozero using pip.
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 49)) 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 77)

- `-m, --monochrome`
  Force monochrome output. See also --color (page 77)

- `-x, --xyz`
  Open pinout.xyz 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:

```
| ooooooooooo000000000000 J8 +====
| 10000000000000000000000000  |   USB
| +====
| Pi Model 3B V1.1  |   |
| +-----+   +====
| |D| |SoC|  |   |   USB
| |S| |   |   +====
| |I| +-----+ |
| | |C|   +====
| | |S|   |   Net
| pwr |HDMI| |I| |A|   +=====
'--' |-----|   |----|V|-----'
```

Revision : a02082
SoC      : BCM2837
RAM      : 1024Mb
Storage  : MicroSD

(continues on next page)

56 https://pinout.xyz/
USB ports: 4 (excluding power)
Ethernet ports: 1
Wi-fi: True
Bluetooth: True
Camera ports (CSI): 1
Display ports (DSI): 1

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

By default, if stdout is a console that supports color, ANSI codes will be used to produce color output. Output can be forced to be `--monochrome` (page 77):

```
$ pinout --monochrome
```

Or forced to be `--color` (page 77), 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 77). The tool understands both old-style revision codes\(^{57}\) (such as for the model B):

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

Or new-style revision codes\(^{58}\) (such as for the Pi Zero W):

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

\(^{57}\) https://www.raspberrypi.org/documentation/hardware/raspberrypi/revision-codes/README.md
\(^{58}\) https://www.raspberrypi.org/documentation/hardware/raspberrypi/revision-codes/README.md
You can also use the tool with *Configuring Remote GPIO* (page 49) to query remote Raspberry Pi’s:

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

Or run the tool directly on a PC using the mock pin implementation (although in this case you’ll almost certainly want to specify the Pi revision manually):
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 lgpio, then pigpio, and finally uses a native Python implementation. Valid values include “rpigpio”, “lgpio”, “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 127) on:

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

And it does, if you’re using the Python or IPython shell, or the IDLE, Thonny or Mu editors. 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()`\(^{59}\) 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)
```

---

\(^{59}\) https://docs.python.org/3.7/library/signal.html#signal.pause
9.2 What’s the difference between when_pressed, is_pressed and wait_for_press?

gpiozero provides a range of different approaches to reading input devices. Sometimes you want to ask if a button’s pressed, sometimes you want to do something until it’s pressed, and sometimes you want something to happen when it’s been pressed, regardless of what else is going on.

In a simple example where the button is the only device in play, all of the options would be equally effective. But as soon as you introduce an extra element, like another GPIO device, you might need to choose the right approach depending on your use case.

- **is_pressed** (page 109) is an attribute which reveals whether the button is currently pressed by returning True or False:

```python
while True:
    if btn.is_pressed:
        print("Pressed")
    else:
        print("Not pressed")
```

- **wait_for_press()** (page 108) as a method which blocks the code from continuing until the button is pressed. Also see **wait_for_release()** (page 108):

```python
while True:
    print("Released. Waiting for press..")
    btn.wait_for_press()
    print("Pressed. Waiting for release...")
    btn.wait_for_release()
```

- **when_pressed** (page 109) is an attribute which assigns a callback function to the event of the button being pressed. Every time the button is pressed, the callback function is executed in a separate thread. Also see **when_released** (page 109):

```python
def pressed():
    print("Pressed")

def released():
    print("Released")

btn.when_pressed = pressed
btn.when_released = released
```

This pattern of options is common among many devices. All **input devices** (page 107) and **internal devices** (page 191) have **is_active**, **when_activated**, **when_deactivated**, **wait_for_active**, and **wait_for_inactive**, and many provide aliases (such as “pressed” for “activated”).

Also see a more advanced approach in the **Source/Values** (page 65) page.

9.3 My event handler isn’t being called

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

```python
button.when_pressed = hello
pause()
```
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 109), the thing that is assigned is the result of calling the pushed function. Because pushed doesn’t explicitly return anything, the result is None60. Hence this is equivalent to doing:

```
b.when_pressed = None
```

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

```
b.when_pressed = pushed
```

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

**Note:** Note that as of v1.5, setting a callback to None61 when it was previously None62 will raise a CallbackSetToNone (page 252) warning, with the intention of alerting users when callbacks are set to None63 accidentally. However, if this is intentional, the warning can be suppressed. See the warnings64 module for reference.

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

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

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

When GPIO Zero is imported it attempts to find a pin driver by importing them in a preferred order (detailed in API - Pins (page 227)). 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 80) environment variable. For example:

```
60 https://docs.python.org/3.7/library/constants.html#None
61 https://docs.python.org/3.7/library/constants.html#None
62 https://docs.python.org/3.7/library/constants.html#None
63 https://docs.python.org/3.7/library/constants.html#None
64 https://docs.python.org/3.7/library/warnings.html#module-warnings
```
### 9.5 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 1.6.2 (/usr/lib/python3/dist-packages)]
>>> require('gpiozero')[0].version
'1.6.2'
```

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 Raspberry Pi OS:

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

Alternatively, install pip with `get-pip`.

### 9.6 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:

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

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

```bash
$ sudo pip3 install gpiozero
```

or:

65 https://docs.python.org/3.7/library/exceptions.html#ImportError
66 https://docs.python.org/3.7/library/warnings.html#module-warnings
$ sudo pip install gpiozero

9.7 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\(^{68}\) to see if it’s been reported before, or check the commits\(^{69}\) on GitHub since the last release to see if it’s been added. The model determination can be found in `gpiozero/pins/data.py`.

9.8 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 203) 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 203) it is restored to its previous state (LED off, pin low).

Read more about Migrating from RPi.GPIO (page 93).

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

The `Button` (page 107) class provides a `when_held` (page 109) 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 109) property). If you want to set `when_held` (page 109) as well as `when_pressed` (page 109), you’ll notice that both callbacks will fire. Sometimes, this is acceptable, but often you’ll want to only fire the `when_pressed` (page 109) 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 109), and instead use `when_released` (page 109) to work out whether it had been held or just pressed:

---

\(^{68}\) https://github.com/gpiozero/gpiozero/issues

\(^{69}\) https://github.com/gpiozero/gpiozero/commits/master
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()
    btn.was_held = False

def pressed():
    print("button was pressed not held")

btn = Button(2)
btn.when_held = held
btn.when_released = released

9.10 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.

9.11 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:

```python
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
```

**9.12 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 Zero⁷⁰, and has been followed by NetworkZero⁷¹, guizero⁷² 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.

---
⁷¹ https://networkzero.readthedocs.io/en/latest/
⁷² https://lawsie.github.io/guizero/
Chapter 10

Backwards Compatibility

GPIO Zero 2.x is a new major version and thus backwards incompatible changes can be expected. We have attempted to keep these as minimal as reasonably possible while taking advantage of the opportunity to clean up things. This chapter documents breaking changes from version 1.x of the library to 2.x.

10.1 Python 2.x support dropped

By far the biggest and most important change is that the Python 2.x series is no longer supported (in practice, this means Python 2.7 is no longer supported). If your code is not compatible with Python 3, you should follow the porting guide\(^73\) in the Python documentation\(^74\).

As of GPIO Zero 2.0, the lowest supported Python version will be 3.5. This base version may advance with minor releases, but we will make a reasonable best effort not to break compatibility with old Python 3.x versions.

10.2 RPIO pin factory removed

The RPIO pin implementation is unsupported on the Raspberry Pi 2 onwards and hence of little practical use these days. Anybody still relying on RPIO’s stable PWM implementation is advised to try the pigpio pin implementation instead (also supported by GPIO Zero).

10.3 Deprecated pin-factory aliases removed

Several deprecated aliases for pin factories, which could be specified by the `GPIOZERO_PIN_FACTORY` (page 80) environment variable, have been removed:

- “PiGPIOPin” is removed in favour of “pigpio”
- “RPiGPIOPin” is removed in favour of “rpigpio”
- “NativePin” is removed in favour of “native”

\(^73\) https://docs.python.org/3/howto/pyporting.html
\(^74\) https://docs.python.org/3/
In other words, you can no longer use the following when invoking your script:

```
$ GPIOZERO_PIN_FACTORY=PiGPIOPin python3 my_script.py
```

Instead, you should use the following:

```
$ GPIOZERO_PIN_FACTORY=pigpio python3 my_script.py
```

### 10.4 Keyword arguments

Many classes in GPIO Zero 1.x were documented as having keyword-only arguments in their constructors and methods. For example, the `PiLiter` (page 173) was documented as having the constructor:

`PiLiter(*, pwm=False, initial_value=False, pin_factory=None)` implying that all its arguments were keyword only.

However, despite being documented in this manner, this was rarely enforced as it was extremely difficult to do so under Python 2.x without complicating the code-base considerably (Python 2.x lacked the `*` syntax to declare keyword-only arguments; they could only be implemented via `**kwargs` arguments and dictionary manipulation).

In GPIO Zero 2.x, all such arguments are now actually keyword arguments. If your code complied with the 1.x documentation you shouldn’t notice any difference. In other words, the following is still fine:

```
from gpiozero import PiLiter
l = PiLiter(pwm=True)
```

However, if you had omitted the keyword you will need to modify your code:

```
from gpiozero import PiLiter
l = PiLiter(True)  # this will no longer work
```

### 10.5 Robots take Motors, and PhaseEnableRobot is removed

The GPIO Zero 1.x API specified that a `Robot` (page 179) was constructed with two tuples that were in turn used to construct two `Motor` (page 137) instances. The 2.x API replaces this with simply passing in the `Motor` (page 137), or `PhaseEnableMotor` (page 138) instances you wish to use as the left and right wheels.

If your current code uses pins 4 and 14 for the left wheel, and 17 and 18 for the right wheel, it may look like this:

```
from gpiozero import Robot
r = Robot(left=(4, 14), right=(17, 18))
```

This should be changed to the following:

```
from gpiozero import Robot, Motor
r = Robot(left=Motor(4, 14), right=Motor(17, 18))
```

Likewise, if you are currently using `PhaseEnableRobot` your code may look like this:
from gpiozero import PhaseEnableRobot

r = PhaseEnableRobot(left=(4, 14), right=(17, 18))

This should be changed to the following:

from gpiozero import Robot, PhaseEnableMotor

r = Robot(left=PhaseEnableMotor(4, 14),
          right=PhaseEnableMotor(17, 18))

This change came about because the Motor (page 137) class was also documented as having two mandatory parameters (forward and backward) and several keyword-only parameters, including the enable pin. However, enable was treated as a positional argument for the sake of constructing Robot (page 179) which was inconsistent. Furthermore, PhaseEnableRobot was more or less a redundant duplicate of Robot (page 179) but was lacking a couple of features added to Robot (page 179) later (notable “curved” turning).

Although the new API requires a little more typing, it does mean that phase enable robot boards now inherit all the functionality of Robot (page 179) because that’s all they use. Theoretically you could also mix and match regular motors and phase-enable motors although there’s little sense in doing so.

The former functionality (passing tuples to the Robot (page 179) constructor) will remain as deprecated functionality for gpiozero 2.0, but will be removed in a future 2.x release.
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` and `OutputDevice`. 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 229).

### 11.1 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)
```

---

75. [https://pypi.org/project/RPi.GPIO/](https://pypi.org/project/RPi.GPIO/)
76. [https://pypi.org/project/RPi.GPIO/](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`.

### 11.2 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:

[77] 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.RISING):
    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)
In GPIO Zero, you assign the `when_pressed` (page 109) and `when_released` (page 109) properties to set up callbacks on those actions:

```python
from gpiozero import Button

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 109) 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 124) 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 107). Other input devices have similar property and method names. There is commonality in naming at base level, such as `InputDevice.is_active` (page 125), which is aliased in a device class, such as `Button.is_pressed` (page 109) and `LightSensor.light_detected` (page 115).

### 11.3 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 131), `LEDBoard` (page 159), `DistanceSensor` (page 115), `Motor` (page 137) and `Robot` (page 179).

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 115) object, specifying the trigger and echo pins, and you would read the `DistanceSensor.distance` (page 117) property which automatically calculates the distance within the implementation of the class.

The `Motor` (page 137) class controls two output pins to drive the motor forwards or backwards. The `Robot` (page 179) 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 159) class takes an arbitrary number of pins, each controlling a single LED. The resulting `LEDBoard` (page 159) 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 39).

### 11.4 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(101):
    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(101):
    led.value = b / 100.0
    sleep(0.01)
```

`PWMLED` (page 129) has a `blink()` (page 129) method which can be used the same was as `LED` (page 127)’s `blink()` (page 128) method, but its PWM capabilities allow for `fade_in` and `fade_out` options to be provided. There is also the `pulse()` (page 130) 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 137), `Servo` (page 139) and `AngularServo` (page 141) classes for information on those. `Motor` (page 137) and `Robot` (page 179) 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 129) objects.

### 11.5 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

---

78 https://pypi.org/project/RPi.GPIO/
Possible by use of the close() (page 203) method on the device.

Note that cleanup only occurs at the point of normal termination of the script. If the script exits due to a program error, cleanup will not be performed. To ensure that cleanup is performed after an exception is raised, the exception must be handled, for example:

```python
from gpiozero import Button

btn = Button(4)

while True:
    try:
        if btn.is_pressed:
            print("Pressed")
    except KeyboardInterrupt:
        print("Ending program")
```

Read more in the relevant FAQ: What’s the gpiozero equivalent of GPIO.cleanup()? (page 85)

## 11.6 Pi Information

RPi.GPIO provides information about the Pi you’re using. The equivalent in GPIO Zero is the function `pi_info()` (page 221):

```python
>>> 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 221) provides.

## 11.7 More

GPIO Zero provides more than just GPIO device support, it includes some support for SPI devices (page 149) 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 194), `TimeOfDay` (page 192), `PingServer` (page 193) and `LoadAverage` (page 196) 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 229) and Configuring Remote GPIO (page 49) 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 231).

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 65).
11.8 FAQs

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

- *How do I keep my script running?* (page 81)
- *Why do I get PinFactoryFallback warnings when I import gpiozero?* (page 83)
- *What’s the gpiozero equivalent of GPIO.cleanup()?* (page 85)
Contributions to the library are welcome! Here are some guidelines to follow.

12.1 Suggestions

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

12.2 Bugs

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

12.3 Documentation

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

12.4 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 263).

---

79 https://github.com/gpiozero/gpiozero/issues/new
80 https://github.com/gpiozero/gpiozero/issues/new
81 https://github.com/gpiozero/gpiozero/tree/master/docs
Do feel free 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.

12.5 Backwards compatibility

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

12.6 Python 2/3

The library is 100% compatible with both Python 2.7 and Python 3 from version 3.2 onwards. Since Python 2 is now past its end-of-life\footnote{http://legacy.python.org/dev/peps/pep-0373/}, the 1.6.2 release (2021-03-18) is the last to support Python 2.
The main GitHub repository for the project can be found at:

https://github.com/gpiozero/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 127) and follow its heritage backward to DigitalOutputDevice (page 143). Follow that back to OutputDevice (page 146) 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).

13.1 Development installation

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

```
$ sudo apt install lsb-release build-essential git 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:

```
$ cd
$ mkvirtualenv -p /usr/bin/python3 gpiozero
$ workon gpiozero
(gpiozero) $ git clone https://github.com/gpiozero/gpiozero.git
(gpiozero) $ cd gpiozero
(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):

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

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

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

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

```
(gpiozero) $ deactivate
$ rmvirtualenv gpiozero
$ rm -rf ~/gpiozero
```

### 13.2 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 texlive-xetex graphviz inkscape \  
python3-sphinx python3-sphinx-rtd-theme latexmk xindy
```

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

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

The HTML output is written to `build/html` while the PDF output goes to `build/latex`.

### 13.3 Test suite

If you wish to run the GPIO Zero test suite, follow the instructions in `Development installation` (page 103) above and then make the “test” target within the sandbox. You’ll also need to install some pip packages:

```
$ workon gpiozero
(gpiozero) $ pip install coverage mock pytest
(gpiozero) $ cd ~/gpiozero
(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\(^{83}\) 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
```

### 13.4 Mock pins

The test suite largely depends on the existence of the mock pin factory `MockFactory` (page 244), which is also useful for manual testing, for example in the Python shell or another REPL. See the section on **Mock pins** (page 231) in the *API - Pins* (page 227) chapter for more information.

---

\(^{83}\) [https://launchpad.net/~deadsnakes/+archive/ubuntu/ppa](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.

### 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 Button

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

Extends *DigitalInputDevice* (page 121) 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*\(^\text{84}\) in the `Button` (page 107) 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**

\(^\text{84}\) [https://docs.python.org/3.7/library/constants.html#False](https://docs.python.org/3.7/library/constants.html#False)
- **pin** *(int[^5] or str[^6])* – The GPIO pin which the button is connected to. See *Pin Numbering* (page 3) for valid pin numbers. If this is *None[^7]* a *GPIODeviceError* (page 249) will be raised.

- **pull_up** *(bool[^8] or None[^9])* – 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[^10]*, the GPIO pin will be pulled low by default. In this case, connect the other side of the button to 3V3. If *None[^11]*, 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[^12] or None[^13])* – See description under *InputDevice* (page 124) for more information.

- **bounce_time** *(float[^14] or None[^15])* – If *None[^16]* (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[^17])* – The length of time (in seconds) to wait after the button is pushed, until executing the *when_held* (page 109) handler. Defaults to 1.

- **hold_repeat** *(bool[^18])* – If *True[^19]*, the *when_held* (page 109) handler will be repeatedly executed as long as the device remains active, every *hold_time* seconds. If *False[^20]* (the default) the *when_held* (page 109) handler will be only be executed once per hold.

- **pin_factory** *(Factory (page 232) or None[^21])* – See *API - Pins* (page 227) 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[^22] or None[^23])* – Number of seconds to wait before proceeding. If this is *None[^24]* (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[^25] or None[^26])* – Number of seconds to wait before proceeding. If this is *None[^27]* (the default), then wait indefinitely until the device is inactive.
**held_time**
The length of time (in seconds) that the device has been held for. This is counted from the first execution of the `when_held` (page 109) event rather than when the device activated, in contrast to `active_time` (page 205). If the device is not currently held, this is `None`\(^{109}\).

**hold_repeat**
If `True`\(^{110}\), `when_held` (page 109) will be executed repeatedly with `hold_time` (page 109) 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 109) handler. If `hold_repeat` (page 109) is True, this is also the length of time between invocations of `when_held` (page 109).

**is_held**
When `True`\(^{111}\), the device has been active for at least `hold_time` (page 109) seconds.

**is_pressed**
Returns `True`\(^{112}\) if the device is currently active and `False`\(^{113}\) otherwise. This property is usually derived from `value` (page 109). Unlike `value` (page 109), this is always a boolean.

**pin**
The `Pin` (page 233) that the device is connected to. This will be `None`\(^{114}\) if the device has been closed (see the `close()` (page 203) 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`\(^{115}\), 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 109) 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`\(^{116}\) (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 it will be passed as that parameter.

Set this property to `None`\(^{117}\) (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).

---

\(^{109}\) [https://docs.python.org/3.7/library/constants.html#None](https://docs.python.org/3.7/library/constants.html#None)

\(^{110}\) [https://docs.python.org/3.7/library/constants.html#True](https://docs.python.org/3.7/library/constants.html#True)

\(^{111}\) [https://docs.python.org/3.7/library/constants.html#True](https://docs.python.org/3.7/library/constants.html#True)

\(^{112}\) [https://docs.python.org/3.7/library/constants.html#False](https://docs.python.org/3.7/library/constants.html#False)

\(^{113}\) [https://docs.python.org/3.7/library/constants.html#True](https://docs.python.org/3.7/library/constants.html#True)

\(^{114}\) [https://docs.python.org/3.7/library/constants.html#None](https://docs.python.org/3.7/library/constants.html#None)

\(^{115}\) [https://docs.python.org/3.7/library/constants.html#None](https://docs.python.org/3.7/library/constants.html#None)

\(^{116}\) [https://docs.python.org/3.7/library/constants.html#None](https://docs.python.org/3.7/library/constants.html#None)

\(^{117}\) [https://docs.python.org/3.7/library/constants.html#None](https://docs.python.org/3.7/library/constants.html#None)
If the function accepts a single mandatory parameter, the device that deactivated it will be passed as that parameter.

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

### 14.1.2 LineSensor (TRCT5000)

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

Extends SmoothedInputDevice (page 123) 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 249) will be raised.
- **pull_up** (bool or None) – See description under InputDevice (page 124) for more information.
- **active_state** (bool or None) – See description under InputDevice (page 124) 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 124) property, and all appropriate events will be fired.

---

https://docs.python.org/3.7/library/constan...None
https://docs.python.org/3.7/library/functions.html#int
https://docs.python.org/3.7/library/functions.html#str
https://docs.python.org/3.7/library/consts.html#None
https://docs.python.org/3.7/library/consts.html#bool
https://docs.python.org/3.7/library/funcs... None
https://docs.python.org/3.7/library/funcs... None
https://docs.python.org/3.7/library/consts.html#float
https://docs.python.org/3.7/library/funcs... float

---

118 https://docs.python.org/3.7/library/constan...None
119 http://camjam.me/?page_id=1035
120 https://docs.python.org/3.7/library/functions.html#int
121 https://docs.python.org/3.7/library/stdlib... html#str
122 https://docs.python.org/3.7/library/consts... html#None
123 https://docs.python.org/3.7/library/funcs... html#bool
124 https://docs.python.org/3.7/library/consts... html#None
125 https://docs.python.org/3.7/library/funcs... html#bool
126 https://docs.python.org/3.7/library/consts... html#None
127 https://docs.python.org/3.7/library/funcs... html#int
128 https://docs.python.org/3.7/library/funcs... html#float
129 https://docs.python.org/3.7/library/funcs... html#float
• **partial** *(bool)* – When False (the default), the object will not return a value for **is_active** (page 124) 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 232) or None)* – See API - Pins (page 227) for more information (this is an advanced feature which most users can ignore).

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

**Parameters**

- **timeout** *(float 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)`
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.

**pin**
The *Pin* (page 233) that the device is connected to. This will be None if the device has been closed (see the close() (page 203) 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 it 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 it will be passed as that parameter.

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

---

130 https://docs.python.org/3.7/library/functions.html#bool
131 https://docs.python.org/3.7/library/constants.html#False
132 https://docs.python.org/3.7/library/constants.html#True
133 https://docs.python.org/3.7/library/constants.html#None
134 https://docs.python.org/3.7/library/functions.html#float
135 https://docs.python.org/3.7/library/constants.html#None
136 https://docs.python.org/3.7/library/constants.html#None
137 https://docs.python.org/3.7/library/functions.html#float
138 https://docs.python.org/3.7/library/constants.html#None
139 https://docs.python.org/3.7/library/constants.html#None
140 https://docs.python.org/3.7/library/constants.html#None
141 https://docs.python.org/3.7/library/constants.html#None
142 https://docs.python.org/3.7/library/constants.html#None
**14.1.3 MotionSensor (D-SUN PIR)**

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

Extends SmoothedInputDevice (page 123) and represents a passive infra-red (PIR) motion sensor like the sort found in the CamJam #2 EduKit\(^{143}\).

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)
pir.wait_for_motion()
print("Motion detected!")
```

**Parameters**

- **pin** (int\(^{144}\) or str\(^{145}\)) – The GPIO pin which the sensor is connected to. See Pin Numbering (page 3) for valid pin numbers. If this is None\(^{146}\) a GPIODeviceError (page 249) will be raised.
- **pull_up** (bool\(^ {147}\) or None\(^ {148}\)) – See description under InputDevice (page 124) for more information.
- **active_state** (bool\(^ {149}\) or None\(^ {150}\)) – See description under InputDevice (page 124) for more information.
- **queue_len** (int\(^ {151}\)) – 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\(^ {152}\)) – The number of values to read from the device (and append to the internal queue) per second. Defaults to 10.
- **threshold** (float\(^ {153}\)) – 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 124) property, and all appropriate events will be fired.
- **partial** (bool\(^ {154}\)) – When False\(^ {155}\) (the default), the object will not return a value for is_active (page 124) until the internal queue has filled with values. Only set this to True\(^ {156}\) if you require values immediately after object construction.
- **pin_factory** (Factory (page 232) or None\(^ {157}\)) – See API - Pins (page 227) 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 (:obj:`float` or :const:`None`) -- Number of seconds to wait before
        proceeding. If this is :const:`None` (the default), then wait indefinitely
        until the device is active.

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

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

motion_detected
    Returns :const:`True` if the value (page 124) currently exceeds threshold (page 124)
    and :const:`False` otherwise.

pin
    The :class:`Pin` (page 233) that the device is connected to. This will be
    :const:`None` if the device has been closed (see the close() (page 203) 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 it will be passed as that parameter.

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

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

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

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

14.1.4 LightSensor (LDR)

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

    Extends :class:`SmoothedInputDevice` (page 123) 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** *(int\(^{169}\) or str\(^{170}\))* — The GPIO pin which the sensor is attached to. See Pin Numbering (page 3) for valid pin numbers. If this is `None`\(^{171}\) a `GPIODeviceError` (page 249) will be raised.

- **queue_len** *(int\(^{172}\))* — The length of the queue used to store values read from the circuit. This defaults to 5.

- **charge_time_limit** *(float\(^{173}\))* — 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\(^{174}\). You may need to adjust this value for different valued capacitors or LDRs.

- **threshold** *(float\(^{175}\))* — 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\(^{176}\))* — When `False`\(^{177}\) (the default), the object will not return a value for `is_active` (page 124) until the internal queue has filled with values. Only set this to `True`\(^{178}\) if you require values immediately after object construction.

- **pin_factory** *(Factory (page 232) or None\(^{179}\))* — See API - Pins (page 227) 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\(^{180}\) or None\(^{181}\))* — 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.

---

\(^{169}\) https://docs.python.org/3.7/library/functions.html#int

\(^{170}\) https://docs.python.org/3.7/library/stdtypes.html#str

\(^{171}\) https://docs.python.org/3.7/library/constants.html#None

\(^{172}\) https://docs.python.org/3.7/library/functions.html#int

\(^{173}\) https://docs.python.org/3.7/library/functions.html#float

\(^{174}\) http://camjam.me/?page_id=623

\(^{175}\) https://docs.python.org/3.7/library/functions.html#float

\(^{176}\) https://docs.python.org/3.7/library/functions.html#bool

\(^{177}\) https://docs.python.org/3.7/library/constants.html#False

\(^{178}\) https://docs.python.org/3.7/library/constants.html#True

\(^{179}\) https://docs.python.org/3.7/library/constants.html#None

\(^{180}\) https://docs.python.org/3.7/library/functions.html#float

\(^{181}\) https://docs.python.org/3.7/library/constants.html#None

\(^{182}\) https://docs.python.org/3.7/library/constants.html#None
Parameters `timeout` ([`float`][183] or `None`[184]) – Number of seconds to wait before proceeding. If this is `None`[185] (the default), then wait indefinitely until the device is active.

`light_detected`
Returns `True`[186] if the `value` (page 124) currently exceeds `threshold` (page 124) and `False`[187] otherwise.

`pin`
The `Pin` (page 233) that the device is connected to. This will be `None`[188] if the device has been closed (see the `close()` (page 203) 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 it will be passed as that parameter.
Set this property to `None`[189] (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 it will be passed as that parameter.
Set this property to `None`[190] (the default) to disable the event.

### 14.1.5 DistanceSensor (HC-SR04)

```python
class gpiozero.DistanceSensor(echo=None, trigger=None, *, queue_len=9, max_distance=1, threshold_distance=0.3, partial=False, pin_factory=None)
```

Extends `SmoothedInputDevice` (page 123) and represents an HC-SR04 ultrasonic distance sensor, as found in the CamJam #3 EduKit[191].

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

1. Connect the GND pin of the sensor to a ground pin on the Pi.
2. Connect the TRIG pin of the sensor a GPIO pin.
3. Connect one end of a 330Ω resistor to the ECHO pin of the sensor.
4. Connect one end of a 470Ω resistor to the GND pin of the sensor.

---

[183](https://docs.python.org/3.7/library/functions.html#float)
[184](https://docs.python.org/3.7/library/constants.html#None)
[185](https://docs.python.org/3.7/library/constants.html#None)
[186](https://docs.python.org/3.7/library/constants.html#True)
[187](https://docs.python.org/3.7/library/constants.html#False)
[188](https://docs.python.org/3.7/library/constants.html#None)
[189](https://docs.python.org/3.7/library/constants.html#None)
[190](https://docs.python.org/3.7/library/constants.html#None)
[191](http://camjam.me/?page_id=1035)
5. Connect the free ends of both resistors to another GPIO pin. This forms the required voltage divider\footnote{https://en.wikipedia.org/wiki/Voltage_divider}. 

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\footnote{https://en.wikipedia.org/wiki/Voltage_divider} given above will actually output ~3V (rather than 3.3V). A simple 2:3 ratio will give 3.333V which implies you can take three resistors of equal value, use one of them instead of the 330Ω resistor, and two of them in series instead of the 470Ω resistor.

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

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

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

**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 229) for further information.

**Parameters**

- **echo** \(\text{(int}^{194} \text{ or str}^{195})\) – The GPIO pin which the ECHO pin is connected to. See *Pin Numbering* (page 3) for valid pin numbers. If this is \textit{None}^{196} a \text{GPIODeviceError} (page 249) will be raised.

- **trigger** \(\text{(int}^{197} \text{ or str}^{198})\) – The GPIO pin which the TRIG pin is connected to. See *Pin Numbering* (page 3) for valid pin numbers. If this is \textit{None}^{199} a \text{GPIODeviceError} (page 249) will be raised.

- **queue_len** \(\text{int}^{200}\) – The length of the queue used to store values read from the sensor. This defaults to 9.

- **max_distance** \(\text{float}^{201}\) – The value (page 117) 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 124) 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 232) or None) – See API - Pins (page 227) 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 117).

**echo**

Returns the **Pin** (page 233) that the sensor’s echo is connected to. This is simply an alias for the usual **pin** (page 126) 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 124) attribute. When **distance** (page 117) is equal to **max_distance** (page 117), **value** (page 124) will be 1.

**threshold_distance**

The distance, measured in meters, that will trigger the **when_in_range** (page 117) and **when_out_of_range** (page 118) events when crossed. This is simply a meter-scaled variant of the usual **threshold** (page 124) attribute.

**trigger**

Returns the **Pin** (page 233) 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.

---

References:

- float: https://docs.python.org/3.7/library/functions.html#float
- bool: https://docs.python.org/3.7/library/functions.html#bool
- False: https://docs.python.org/3.7/library/constants.html#False
- True: https://docs.python.org/3.7/library/constants.html#True
- None: https://docs.python.org/3.7/library/constants.html#None
- float: https://docs.python.org/3.7/library/functions.html#float
- None: https://docs.python.org/3.7/library/constants.html#None
- None: https://docs.python.org/3.7/library/constants.html#None

---

14.1. Regular Classes

117
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 it will be passed as that parameter.

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

**when_out_of_range**

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

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

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

### 14.1.6 RotaryEncoder

```python
class gpiozero.RotaryEncoder(a, b, *, bounce_time=None, max_steps=16, thresh-old_steps=(0, 0), wrap=False, pin_factory=None)
```

Represents a simple two-pin incremental rotary encoder device.

These devices typically have three pins labelled “A”, “B”, and “C”. Connect A and B directly to two GPIO pins, and C (“common”) to one of the ground pins on your Pi. Then simply specify the A and B pins as the arguments when constructing this class.

For example, if your encoder’s A pin is connected to GPIO 21, and the B pin to GPIO 20 (and presumably the C pin to a suitable GND pin), while an LED (with a suitable 300Ω resistor) is connected to GPIO 5, the following session will result in the brightness of the LED being controlled by dialling the rotary encoder back and forth:

```python
>>> from gpiozero import RotaryEncoder
>>> from gpiozero.tools import scaled_half
>>> rotor = RotaryEncoder(21, 20)
>>> led = PWMLED(5)
>>> led.source = scaled_half(rotor.values)
```

**Parameters**

- `a (int or str)` – The GPIO pin connected to the “A” output of the rotary encoder.
- `b (int or str)` – The GPIO pin connected to the “B” output of the rotary encoder.
- `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.
- `max_steps (int)` – The number of steps clockwise the encoder takes to change the value (page 120) from 0 to 1, or counter-clockwise from 0 to -1. If this is 0,
then the encoder’s value (page 120) never changes, but you can still read steps (page 119) to determine the integer number of steps the encoder has moved clockwise or counter clockwise.

- **threshold_steps (tuple of int)** – A (min, max) tuple of steps between which the device will be considered “active”, inclusive. In other words, when steps (page 119) is greater than or equal to the min value, and less than or equal the max value, the active property will be True and the appropriate events (when_activated, when_deactivated) will be fired. Defaults to (0, 0).

- **wrap (bool)** – If True and max_steps is non-zero, when the steps (page 119) reaches positive or negative max_steps it wraps around by negation. Defaults to False.

- **pin_factory (Factory (page 232) or None)** – See API - Pins (page 227) for more information (this is an advanced feature which most users can ignore).

```python
wait_for_rotate(timeout=None)
```

Pause the script until the encoder is rotated at least one step in either direction, 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 encoder is rotated.

```python
wait_for_rotate_clockwise(timeout=None)
```

Pause the script until the encoder is rotated at least one step clockwise, 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 encoder is rotated clockwise.

```python
wait_for_rotate_counter_clockwise(timeout=None)
```

Pause the script until the encoder is rotated at least one step counter-clockwise, 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 encoder is rotated counter-clockwise.

**max_steps**

The number of discrete steps the rotary encoder takes to move value (page 120) from 0 to 1 clockwise, or 0 to -1 counter-clockwise. In another sense, this is also the total number of discrete states this input can represent.

**steps**

The “steps” value of the encoder starts at 0. It increments by one for every step the encoder is rotated clockwise, and decrements by one for every step it is rotated counter-clockwise.

The steps value is limited by max_steps (page 119). It will not advance beyond positive or

[224](https://docs.python.org/3.7/library/constants.html#True)
[225](https://docs.python.org/3.7/library/functions.html#bool)
[226](https://docs.python.org/3.7/library/constants.html#True)
[227](https://docs.python.org/3.7/library/constants.html#False)
[228](https://docs.python.org/3.7/library/constants.html#None)
[229](https://docs.python.org/3.7/library/functions.html#float)
[230](https://docs.python.org/3.7/library/constants.html#None)
[231](https://docs.python.org/3.7/library/constants.html#None)
[232](https://docs.python.org/3.7/library/functions.html#float)
[233](https://docs.python.org/3.7/library/constants.html#None)
[234](https://docs.python.org/3.7/library/constants.html#None)
[235](https://docs.python.org/3.7/library/functions.html#float)
[236](https://docs.python.org/3.7/library/constants.html#None)
negative `max_steps` (page 119), unless `wrap` (page 120) is `True`\(^\text{238}\) in which case it will roll around by negation. If `max_steps` (page 119) is zero then steps are not limited at all, and will increase infinitely in either direction, but `value` (page 120) will return a constant zero.

Note that, in contrast to most other input devices, because the rotary encoder has no absolute position the `steps` (page 119) attribute (and `value` (page 120) by corollary) is writable.

### `threshold_steps`

The minimum and maximum number of steps between which `is_active` will return `True`\(^\text{239}\). Defaults to (0, 0).

### `value`

Represents the value of the rotary encoder as a value between -1 and 1. The value is calculated by dividing the value of `steps` (page 119) into the range from negative `max_steps` (page 119) to positive `max_steps` (page 119).

Note that, in contrast to most other input devices, because the rotary encoder has no absolute position the `value` (page 120) attribute is writable.

### `when_rotated`

The function to be run when the encoder is rotated in either direction.

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`\(^\text{240}\) (the default) to disable the event.

### `when_rotated_clockwise`

The function to be run when the encoder is rotated clockwise.

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`\(^\text{241}\) (the default) to disable the event.

### `when_rotated_counter_clockwise`

The function to be run when the encoder is rotated counter-clockwise.

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`\(^\text{242}\) (the default) to disable the event.

### `wrap`

If `True`\(^\text{243}\), when `value` (page 120) reaches its limit (-1 or 1), it “wraps around” to the opposite limit. When `False`\(^\text{244}\), the value (and the corresponding `steps` (page 119) attribute) simply don’t advance beyond their limits.

\(^{238}\) [https://docs.python.org/3.7/library/constants.html#True](https://docs.python.org/3.7/library/constants.html#True)

\(^{239}\) [https://docs.python.org/3.7/library/constants.html#True](https://docs.python.org/3.7/library/constants.html#True)

\(^{240}\) [https://docs.python.org/3.7/library/constants.html#None](https://docs.python.org/3.7/library/constants.html#None)

\(^{241}\) [https://docs.python.org/3.7/library/constants.html#None](https://docs.python.org/3.7/library/constants.html#None)

\(^{242}\) [https://docs.python.org/3.7/library/constants.html#None](https://docs.python.org/3.7/library/constants.html#None)

\(^{243}\) [https://docs.python.org/3.7/library/constants.html#True](https://docs.python.org/3.7/library/constants.html#True)

\(^{244}\) [https://docs.python.org/3.7/library/constants.html#False](https://docs.python.org/3.7/library/constants.html#False)
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):

```
+---------------------+                  +---------------------+
| GPIODevice          |                  | DistanceSensor       |
| SmoothedInputDevice |                  | LightSensor          |
| InputDevice         |                  | LineSensor           |
+---------------------+                  +---------------------+
| Button              |                  | MotionSensor         |
| DigitalInputDevice  |                  |                     |
|                     |                  |                     |
+---------------------+                  +---------------------+
```

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

14.2.1 DigitalInputDevice

```python
class gpiozero.DigitalInputDevice(pin=None, *, 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 124) with machinery to fire the active and inactive events for devices that operate in a typical digital manner: straight forward on / off states with (reasonably) clean transitions between the two.

**Parameters**

- `pin (int | 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 249) will be raised.
- `pull_up (bool | None)` – See description under `InputDevice` (page 124) for more information.
- `active_state (bool | None)` – See description under `InputDevice` (page 124) for more information.

245 https://docs.python.org/3.7/library/functions.html#int
246 https://docs.python.org/3.7/library/stdtypes.html#str
247 https://docs.python.org/3.7/library/constants.html#None
248 https://docs.python.org/3.7/library/functions.html#bool
249 https://docs.python.org/3.7/library/constants.html#None
250 https://docs.python.org/3.7/library/constants.html#None
251 https://docs.python.org/3.7/library/constants.html#None
• **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 232) or None)* — See **API - Pins** (page 227) 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 it 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 it will be passed as that parameter.

---

252 https://docs.python.org/3.7/library/functions.html#float
253 https://docs.python.org/3.7/library/constants.html#None
254 https://docs.python.org/3.7/library/constants.html#None
255 https://docs.python.org/3.7/library/constants.html#None
256 https://docs.python.org/3.7/library/functions.html#float
257 https://docs.python.org/3.7/library/constants.html#None
258 https://docs.python.org/3.7/library/constants.html#None
259 https://docs.python.org/3.7/library/constants.html#None
260 https://docs.python.org/3.7/library/constants.html#None
261 https://docs.python.org/3.7/library/constants.html#None
262 https://docs.python.org/3.7/library/constants.html#None
263 https://docs.python.org/3.7/library/constants.html#None
264 https://docs.python.org/3.7/library/constants.html#None

Chapter 14. API - Input Devices
Set this property to `None` (the default) to disable the event.

### 14.2.2 SmoothedInputDevice

```python
class gpiozero.SmoothedInputDevice(
    pin=None, *, pull_up=False, active_state=None,
    threshold=0.5, queue_len=5, sample_wait=0.0, partial=False,
    average=<function median>, ignore=None,
    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 124) 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 124) property.

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

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

**Parameters**

- `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 249) will be raised.
- `pull_up (bool or None)` – See description under `InputDevice` (page 124) for more information.
- `active_state (bool or None)` – See description under `InputDevice` (page 124) 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 124) 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` or `None`) – See *API - Pins* (page 227) for more information (this is an advanced feature which most users can ignore).

    `is_active`  
    Returns True if the value currently exceeds `threshold` and False otherwise.

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

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

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

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

### 14.2.3 InputDevice

**class gpiozero.InputDevice**(pin=None, *, pull_up=False, active_state=None, pin_factory=None)  
Represents a generic GPIO input device.

This class extends *GPIODevice* (page 125) 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 125) property is adjusted accordingly so that True still means active regardless of the `pull_up` setting.

**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 249) will be raised.

---

278 https://docs.python.org/3.7/library/constants.html#True  
279 https://docs.python.org/3.7/library/statistics.html#statistics.median  
280 https://docs.python.org/3.7/library/stdtypes.html#frozenset  
281 https://docs.python.org/3.7/library/constants.html#None  
282 https://docs.python.org/3.7/library/constants.html#None  
283 https://docs.python.org/3.7/library/constants.html#True  
284 https://docs.python.org/3.7/library/constants.html#False  
285 https://docs.python.org/3.7/library/constants.html#False  
286 https://docs.python.org/3.7/library/constants.html#True  
287 https://docs.python.org/3.7/library/constants.html#True  
288 https://docs.python.org/3.7/library/constants.html#None  
289 https://docs.python.org/3.7/library/functions.html#int  
290 https://docs.python.org/3.7/library/stdtypes.html#str  
291 https://docs.python.org/3.7/library/constants.html#None
• **pull_up** *(bool or None)* – If True, the pin will be pulled high with an internal resistor. If False (the default), the pin will be pulled low. If None, 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 or None)* – If True, when the hardware pin state is HIGH, the software pin is HIGH. If False, 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 or None)* – See API - Pins (page 227) 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 125). Unlike value (page 125), 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.

### 14.2.4 GPIODevice

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

Extends Device (page 203). Represents a generic GPIO device and provides the services common to all single-pin GPIO devices (like ensuring two GPIO devices do not share a pin (page 126)).

**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 249) will be raised. If the pin is already in use by another device, GPIOPinInUse (page 249) will be raised.

**close()**

Shut down the device and release all associated resources (such as GPIO pins).

This method is idempotent (can be called on an already closed device without any side-effects). It is primarily intended for interactive use at the command line. It disables the device and releases its pin(s) for use by another device.

---

292 https://docs.python.org/3.7/library/functions.html#bool
293 https://docs.python.org/3.7/library/constants.html#None
294 https://docs.python.org/3.7/library/constants.html#True
295 https://docs.python.org/3.7/library/constants.html#False
296 https://docs.python.org/3.7/library/constants.html#bool
297 https://docs.python.org/3.7/library/functions.html#bool
298 https://docs.python.org/3.7/library/constants.html#None
299 https://docs.python.org/3.7/library/constants.html#None
300 https://docs.python.org/3.7/library/constants.html#None
301 https://docs.python.org/3.7/library/constants.html#None
302 https://docs.python.org/3.7/library/functions.html#int
303 https://docs.python.org/3.7/library/functions.html#str
304 https://docs.python.org/3.7/library/constants.html#None
305 https://docs.python.org/3.7/library/constants.html#None
306 https://docs.python.org/3.7/library/constants.html#None
307 https://docs.python.org/3.7/library/constants.html#None
308 https://docs.python.org/3.7/library/constants.html#None
309 https://docs.python.org/3.7/library/constants.html#None
310 https://docs.python.org/3.7/library/constants.html#None
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 203) 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 125) 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 233) that the device is connected to. This will be None if the device has been closed (see the close() (page 203) 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.

---

311 https://docs.python.org/3.7/reference/compound_stmts.html#with
312 https://docs.python.org/3.7/library/constants.html#True
313 https://docs.python.org/3.7/library/constants.html#None
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.

15.1 Regular Classes

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

15.1.1 LED

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

Parameters

- **pin** *(int314 or str315)* – The GPIO pin which the LED is connected to. See *Pin Numbering* (page 3) for valid pin numbers. If this is None316 a *GPIODeviceError*

314 https://docs.python.org/3.7/library/functions.html#int
315 https://docs.python.org/3.7/library/stdtypes.html#str
316 https://docs.python.org/3.7/library/constants.html#None
gpiozero 1.6.1 Documentation

(page 249) will be raised.

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

- **initial_value** *(bool or None)* - If `False` (the default), the LED will be off initially. If `None`, the LED will be left in whatever state the pin is found in when configured for output (warning: this can be on). If `True`, the LED will be switched on initially.

- **pin_factory** *(Factory or None)* - See API - Pins (page 227) 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)* - 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).

```
off()
```

Turns the device off.

```
on()
```

Turns the device on.

```
toggle()
```

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

**is_lit**

Returns `True` if the device is currently active and `False` otherwise. This property is usually derived from `value` (page 129). Unlike `value` (page 129), this is always a boolean.

**pin**

The `Pin` (page 233) that the device is connected to. This will be `None` if the device has been closed (see the `close()` (page 203) method). When dealing with GPIO pins, query `pin.number` to discover the GPIO pin (in BCM numbering) that the device is connected to.

---

317 https://docs.python.org/3.7/library/functions.html#bool
318 https://docs.python.org/3.7/library/constants.html#True
319 https://docs.python.org/3.7/library/constants.html#False
320 https://docs.python.org/3.7/library/functions.html#bool
321 https://docs.python.org/3.7/library/constants.html#None
322 https://docs.python.org/3.7/library/constants.html#None
323 https://docs.python.org/3.7/library/constants.html#None
324 https://docs.python.org/3.7/library/constants.html#None
325 https://docs.python.org/3.7/library/constants.html#None
326 https://docs.python.org/3.7/library/functions.html#float
327 https://docs.python.org/3.7/library/functions.html#float
328 https://docs.python.org/3.7/library/functions.html#int
329 https://docs.python.org/3.7/library/functions.html#int
330 https://docs.python.org/3.7/library/constants.html#None
331 https://docs.python.org/3.7/library/constants.html#None
332 https://docs.python.org/3.7/library/constants.html#True
333 https://docs.python.org/3.7/library/constants.html#True
334 https://docs.python.org/3.7/library/constants.html#False
335 https://docs.python.org/3.7/library/constants.html#False
336 https://docs.python.org/3.7/library/constants.html#None

Chapter 15. API - Output Devices
value
Returns 1 if the device is currently active and 0 otherwise. Setting this property changes the
state of the device.

15.1.2 PWMLED

class gpiozero.PWMLED(pin=None, *, active_high=True, initial_value=0, frequency=100,
pin_factory=None)
Extends PWMOutputDevice (page 144) and represents a light emitting diode (LED) with variable
brightness.

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

Parameters
- `pin (int or str)` – The GPIO pin which the LED is connected to. See Pin
  Numbering (page 3) for valid pin numbers. If this is None a
  GPIODeviceError (page 249) 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 (float)` – If 0 (the default), the LED will be off initially.
  Other values between 0 and 1 can be specified as an initial brightness for the
  LED. Note that None cannot be specified (unlike the parent class) as there is
  no way to tell PWM not to alter the state of the pin.
- `frequency (int)` – The frequency (in Hz) of pulses emitted to drive the LED.
  Defaults to 100Hz.
- `pin_factory (Factory or None)` – See API - Pins (page 227) 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.
• `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 130) is 0.0), this changes it to “fully” on (`value` (page 130) is 1.0). If the device has a duty cycle (`value` (page 130)) of 0.1, this will toggle it to 0.9, and so on.

`is_lit`
Returns True if the device is currently active (`value` (page 130) is non-zero) and False otherwise.

`pin`
The Pin (page 233) that the device is connected to. This will be None if the device has been closed (see the `close()` (page 203) method). When dealing with GPIO pins, query `pin.number` to discover the GPIO pin (in BCM numbering) that the device is connected to.

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

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

Extends `Device` (page 203) 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 (int or str)` – The GPIO pin that controls the red component of the RGB LED. See `Pin Numbering` (page 3) for valid pin numbers. If this is `None` a `GPIODeviceError` (page 249) will be raised.
- `green (int or str)` – The GPIO pin that controls the green component of the RGB LED.
- `blue (int or str)` – The GPIO pin that controls the blue component of the RGB LED.
- `active_high (bool)` – Set to `True` (the default) for common cathode RGB LEDs. If you are using a common anode RGB LED, set this to `False`.
- `initial_value (Color or tuple)` – The initial color for the RGB LED. Defaults to black `(0, 0, 0)`.
- `pwm (bool)` – If `True` (the default), construct `PWMLED` (page 129) instances for each component of the RGBLED. If `False`, construct regular `LED` (page 127) instances, which prevents smooth color graduations.

---

368 https://colorzero.readthedocs.io/
369 https://docs.python.org/3.7/library/functions.html#int
370 https://docs.python.org/3.7/library/stdtypes.html#str
371 https://docs.python.org/3.7/library/constan...None
372 https://docs.python.org/3.7/library/functions.html#int
373 https://docs.python.org/3.7/library/stdtypes.html#str
374 https://docs.python.org/3.7/library/functions.html#int
375 https://docs.python.org/3.7/library/stdtypes.html#str
376 https://docs.python.org/3.7/library/functions.html#bool
377 https://docs.python.org/3.7/library/constan...True
378 https://docs.python.org/3.7/library/constan...False
380 https://docs.python.org/3.7/library/stdtypes.html#tuple
381 https://docs.python.org/3.7/library/functions.html#bool
382 https://docs.python.org/3.7/library/constan...True
383 https://docs.python.org/3.7/library/constan...False

15.1. Regular Classes 131
• **pin_factory** *(Factory (page 232) or None)* – See *API - Pins* (page 227) for more information (this is an advanced feature which most users can ignore).

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

Make the device turn on and off repeatedly.

**Parameters**

- **on_time** *(float)* – 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).

```python
toggle()
```

Toggles the state of the device. If the device is currently off (value (page 133) is `(0, 0, 0)`), this changes it to “fully” on (value (page 133) is `(1, 1, 1)`). If the device has a specific color, this method inverts the color.

```python
blue
```

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

```python
color
```

Represents the color of the LED as a Color object.

```python
green
```

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

```python
is_lit
```

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

```python
red
```

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

```python
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)`.
15.1.4 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](#) for valid pin numbers. If this is None a `GPIODeviceError` 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](#) for more information (this is an advanced feature which most users can ignore).

```python
bz.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).

```
off()
```

Turns the device off.

```
on()
```

Turns the device on.

```
toggle()
```

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

```
is_active
```

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

```
pin
```

The `Pin` (page 233) that the device is connected to. This will be **None** if the device has been closed (see the `close()` (page 203) 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.

### 15.1.5 TonalBuzzer

class `gpiozero.TonalBuzzer`(*pin=None, *, initial_value=None, mid_tone=<Tone note='A4', midi=69 frequency=440.00Hz>, octaves=1, pin_factory=None)

Extends `CompositeDevice` (page 188) and represents a tonal buzzer.

**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 249) will be raised.

- **initial_value** (float) – If **None** (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** (int or str) – The tone which is represented the device’s middle value (0). The default is “A4” (MIDI note 69).
• **octaves** *(int)* – 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** *(Factory (page 232) or None)* – See API - Pins (page 227) 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 242).

**play**(tone)

Play the given tone. This can either be an instance of *Tone* (page 219) or can be anything that could be used to construct an instance of *Tone* (page 219).

For example:

```python
>>> 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)
```

**stop()**

Turn the buzzer off. This is equivalent to setting *value* (page 136) 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 136) is 1.

**mid_tone**

The middle tone available, i.e. the tone played when *value* (page 136) is 0.

**min_tone**

The lowest tone that the buzzer can play, i.e. the tone played when *value* (page 136) is -1.

**octaves**

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

**tone**

Returns the *Tone* (page 219) 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.

---

452 https://docs.python.org/3.7/library/functions.html#int
453 https://docs.python.org/3.7/library/constants.html#None
454 https://docs.python.org/3.7/library/constants.html#None
455 https://docs.python.org/3.7/library/constants.html#True
456 https://docs.python.org/3.7/library/constants.html#False
457 https://docs.python.org/3.7/library/constants.html#None
458 https://docs.python.org/3.7/library/constants.html#None
15.1.6 Motor

class gpiozero.Motor(forward, backward, *, enable=None, pwm=True, pin_factory=None)
Extends CompositeDevice (page 188) and represents a generic motor connected to a bi-directional
motor driver circuit (i.e. an H-bridge).159

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”:

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

Parameters

- **forward** (`int` or `str`) – 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 a GPIODeviceError (page 249) will be raised.

- **backward** (`int` or `str`) – 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 a GPIODeviceError (page 249) will be raised.

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

- **pwm** (`bool`) – If True (the default), construct PWMOutputDevice (page 144)
  instances for the motor controller pins, allowing both direction and variable speed
  control. If False, construct DigitalOutputDevice (page 143) instances, al-
  lowing only direction control.

- **pin_factory** (`Factory` or None) – See API - Pins (page 227) for
  more information (this is an advanced feature which most users can ignore).

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

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

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

159 https://en.wikipedia.org/wiki/H_bridge
160 https://en.wikipedia.org/wiki/H_bridge
161 https://docs.python.org/3.7/library/functions.html#int
162 https://docs.python.org/3.7/library/stdtypes.html#str
163 https://docs.python.org/3.7/library/constants.html#None
164 https://docs.python.org/3.7/library/functions.html#int
165 https://docs.python.org/3.7/library/stdtypes.html#str
166 https://docs.python.org/3.7/library/constants.html#None
167 https://docs.python.org/3.7/library/functions.html#int
168 https://docs.python.org/3.7/library/stdtypes.html#str
169 https://docs.python.org/3.7/library/constants.html#None
170 https://docs.python.org/3.7/library/functions.html#bool
171 https://docs.python.org/3.7/library/constants.html#True
172 https://docs.python.org/3.7/library/constants.html#False
173 https://docs.python.org/3.7/library/constants.html#None
174 https://docs.python.org/3.7/library/functions.html#float
Parameters speed (float) – The speed at which the motor should turn. Can be any value between 0 (stopped) and the default 1 (maximum speed) if pwm was True when the class was constructed (and only 0 or 1 if not).

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), with 0 representing stopped.

15.1.7 PhaseEnableMotor

class gpiozero.PhaseEnableMotor(phase, enable, *, pwm=True, pin_factory=None)
Extends CompositeDevice (page 188) 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 249) 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 249) will be raised.

- **pwm** (bool) – If True (the default), construct PWMOutputDevice (page 144) instances for the motor controller pins, allowing both direction and variable speed control. If False, construct DigitalOutputDevice (page 143) instances, allowing only direction control.

- **pin_factory** (Factory or None) – See API - Pins (page 227) 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).

15.1.8 Servo

class gpiozero.Servo(pin=None, *, initial_value=0.0, min_pulse_width=0.001, max_pulse_width=0.002, frame_width=0.02, pin_factory=None)
    Extends CompositeDevice and represents a PWM-controlled servo motor connected to a GPIO pin.

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

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

```
from gpiozero import Servo
from time import sleep

servo = Servo(17)

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

You can also use the value (page 141) property to move the servo to a particular position, on a scale from -1 (min) to 1 (max) where 0 is the mid-point:

- https://docs.python.org/3.7/library/functions.html#float
- https://docs.python.org/3.7/library/functions.html#float
- https://docs.python.org/3.7/library/constants.html#True
- https://docs.python.org/3.7/library/constants.html#False
from gpiozero import Servo

servo = Servo(17)
servo.value = 0.5

Note: To reduce servo jitter, use the pigpio pin driver rather than the default RPi.GPIO driver (pigpio uses DMA sampling for much more precise edge timing). See Changing the pin factory (page 229) for further information.

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 249) 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 141)).
- **min_pulse_width** (`float`) – The pulse width corresponding to the servo’s minimum position. This defaults to 1ms.
- **max_pulse_width** (`float`) – The pulse width corresponding to the servo’s maximum position. This defaults to 2ms.
- **frame_width** (`float`) – The length of time between servo control pulses measured in seconds. This defaults to 20ms which is a common value for servos.
- **pin_factory** (`Factory` (page 232) or `None`) – See API - Pins (page 227) for more information (this is an advanced feature which most users can ignore).

**detach()**

Temporarily disable control of the servo. This is equivalent to setting `value` (page 141) to `None`.

**max()**

Set the servo to its maximum position.

**mid()**

Set the servo to its mid-point position.

**min()**

Set the servo to its minimum position.

**frame_width**

The time between control pulses, measured in seconds.

**is_active**

Composite devices are considered “active” if any of their constituent devices have a “truthy” value.
max_pulse_width
The control pulse width corresponding to the servo’s maximum position, measured in seconds.

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

pulse_width
Returns the current pulse width controlling the servo.

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.

15.1.9 AngularServo
class gpiozero.AngularServo(pin=None, *, initial_angle=0.0, min_angle=-90, max_angle=90, min_pulse_width=0.001, max_pulse_width=0.002, frame_width=0.02, pin_factory=None)
Extends Servo (page 139) 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 139) 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
```

You should now be able to construct an AngularServo (page 141) instance with the correct bounds:

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

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

Parameters

\[^{504}\](https://docs.python.org/3.7/library/constants.html#None)
• `pin(int505 or str506)` – The GPIO pin that the servo is connected to. See Pin Numbering (page 3) for valid pin numbers. If this is None507 a GPIODeviceError (page 249) will be raised.

• `initial_angle(float508)` – 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. None509 means to start the servo un-controlled (see value (page 143)).

• `min_angle(float510)` – 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(float511)` – 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(float512)` – The pulse width corresponding to the servo’s minimum position. This defaults to 1ms.

• `max_pulse_width(float513)` – The pulse width corresponding to the servo’s maximum position. This defaults to 2ms.

• `frame_width(float514)` – 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 232) or None515)` – See API - Pins (page 227) for more information (this is an advanced feature which most users can ignore).

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

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

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

`angle`  
The position of the servo as an angle measured in degrees. This will only be accurate if `min_angle` (page 142) and `max_angle` (page 142) have been set appropriately in the constructor.

This can also be the special value None516 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 `max()` (page 142) is called.

`min_angle`  
The minimum angle that the servo will rotate to when `min()` (page 142) is called.

---

505 https://docs.python.org/3.7/library/functions.html#int  
506 https://docs.python.org/3.7/library/stdtypes.html#str  
507 https://docs.python.org/3.7/library//constants.html#None  
508 https://docs.python.org/3.7/library/functions.html#float  
509 https://docs.python.org/3.7/library/constants.html#None  
510 https://docs.python.org/3.7/library/functions.html#float  
511 https://docs.python.org/3.7/library/constants.html#None  
512 https://docs.python.org/3.7/library/functions.html#float  
513 https://docs.python.org/3.7/library/functions.html#float  
514 https://docs.python.org/3.7/library/functions.html#float  
515 https://docs.python.org/3.7/library/constants.html#None  
516 https://docs.python.org/3.7/library/constants.html#None
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.

15.2 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):

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

15.2.1 DigitalOutputDevice

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

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

This class extends `OutputDevice` (page 146) with a `blink()` (page 144) method which uses an optional background thread to handle toggling the device state without further interaction.

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 249) will be raised.

---

517 https://docs.python.org/3.7/library/constants.html#None
518 https://docs.python.org/3.7/library/functions.html#int
519 https://docs.python.org/3.7/library/stdtypes.html#str
520 https://docs.python.org/3.7/library/constants.html#None
• `active_high` *(bool)* – If True (the default), the `on()` (page 144) method will set the GPIO to HIGH. If False, the `on()` (page 144) method will set the GPIO to LOW (the `off()` (page 144) 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 (page 232) or None)* – See API - Pins (page 227) 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)* – 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).

`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.

### 15.2.2 PWMOutputDevice

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

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

**Parameters**

---

521 https://docs.python.org/3.7/library/functions.html#bool
522 https://docs.python.org/3.7/library/constant.html#False
523 https://docs.python.org/3.7/library/constant.html#True
524 https://docs.python.org/3.7/library/functions.html#bool
525 https://docs.python.org/3.7/library/constant.html#None
530 https://docs.python.org/3.7/library/constant.html#float
531 https://docs.python.org/3.7/library/functions.html#float
532 https://docs.python.org/3.7/library/functions.html#int
533 https://docs.python.org/3.7/library/constant.html#None
534 https://docs.python.org/3.7/library/constant.html#None
535 https://docs.python.org/3.7/library/functions.html#bool
536 https://docs.python.org/3.7/library/constant.html#True
537 https://docs.python.org/3.7/library/constant.html#False
• **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 249) will be raised.

• **active_high** *(bool)* – If `True` (the default), the `on()` (page 145) method will set the GPIO to HIGH. If `False`, the `on()` (page 145) method will set the GPIO to LOW (the `off()` (page 145) 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 227) for more information (this is an advanced feature which most users can ignore).

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

Make the device turn on and off repeatedly.

**Parameters**

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

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

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

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

• **n** *(int 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.

```python
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 146) is 0.0), this changes it to “fully” on (value (page 146) is 1.0). If the device has a duty cycle (value (page 146)) 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 146) 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.

15.2.3 OutputDevice

class gpizero.OUTPUT_DEVICE(pin=None, *args, active_high=True, initial_value=False, pin_factory=None)

Represents a generic GPIO output device.

This class extends GPIODevice (page 125) to add facilities common to GPIO output devices: an on() (page 147) method to switch the device on, a corresponding off() (page 147) method, and a toggle() (page 147) 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 249) will be raised.
• *active_high* (*bool*571) – If *True*572 (the default), the *on()* (page 147) method will set the GPIO to HIGH. If *False*573, the *on()* (page 147) method will set the GPIO to LOW (the *off()* (page 147) method always does the opposite).

• *initial_value* (*bool*574 or *None*575) – If *False*576 (the default), the device will be off initially. If *None*577, the device will be left in whatever state the pin is found in when configured for output (warning: this can be on). If *True*578, the device will be switched on initially.

• *pin_factory* (*Factory* (page 232) or *None*579) – See *API - Pins* (page 227) 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*580, the *value* (page 147) property is *True*581 when the device’s *pin* (page 126) is high. When *False*582 the *value* (page 147) property is *True*583 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 147) (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.

15.2.4 GPIODevice

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

Extends *Device* (page 203). 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 126)).

**Parameters** *pin* (*int*584 or *str*585) – The GPIO pin that the device is connected to. See *Pin Numbering* (page 3) for valid pin numbers. If this is *None*586 a *GPIODeviceError* (page 249) will be raised. If the pin is already in use by another device, *GPIOPinInUse* (page 249) will be raised.

**close()**

Shut down the device and release all associated resources (such as GPIO pins).
This method is idempotent (can be called on an already closed device without any side-effects). It 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 203) 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 125) 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 233) that the device is connected to. This will be `None` if the device has been closed (see the `close()` (page 203) method). When dealing with GPIO pins, query `pin.number` to discover the GPIO pin (in BCM numbering) that the device is connected to.

value

Returns a value representing the device’s state. Frequently, this is a boolean value, or a number between 0 and 1 but some devices use larger ranges (e.g. -1 to +1) and composite devices usually use tuples to return the states of all their subordinate components.
SPI stands for Serial Peripheral Interface\(^{590}\) 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.

### 16.1 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).

\(^{590}\) https://en.wikipedia.org/wiki/Serial_Peripheral_Interface_Bus
You cannot mix these two schemes, i.e. attempting to specify `port` and `clock_pin` will result in `SPIBadArgs` (page 249) 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 227) 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.

## 16.2 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).

### 16.2.1 MCP3001

```python
class gpiozero.MCP3001(max_voltage=3.3, **spi_args)
The MCP3001\(^{591}\) 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).
```

### 16.2.2 MCP3002

```python
class gpiozero.MCP3002(channel=0, differential=False, max_voltage=3.3, **spi_args)
The MCP3002\(^{592}\) 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 151) 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).

16.2.3 MCP3004

class gpiozero.MCP3004(channel=0, differential=False, max_voltage=3.3, **spi_args)
The MCP3004\(^{593}\) 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 151) 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).

16.2.4 MCP3008

class gpiozero.MCP3008(channel=0, differential=False, max_voltage=3.3, **spi_args)
The MCP3008\(^{594}\) 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 151) 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).

\(^{593}\) http://www.farnell.com/datasheets/808965.pdf

\(^{594}\) http://www.farnell.com/datasheets/808965.pdf

16.2. Analog to Digital Converters (ADC) 151
16.2.5 MCP3201

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

16.2.6 MCP3202

class gpiozero.MCP3202(channel=0, differential=False, max_voltage=3.3, **spi_args)
The MCP3202 is a 12-bit analog to digital converter with 2 channels (0-1).

channel
The channel to read data from. The MCP3008/3208/3304 have 8 channels (0-7), while the MCP3004/3204/3302 have 4 channels (0-3), 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 151) 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).

16.2.7 MCP3204

class gpiozero.MCP3204(channel=0, differential=False, max_voltage=3.3, **spi_args)
The MCP3204 is a 12-bit analog to digital converter with 4 channels (0-3).

channel
The channel to read data from. The MCP3008/3208/3304 have 8 channels (0-7), while the MCP3004/3204/3302 have 4 channels (0-3), 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 151) 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).

16.2.8 MCP3208

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

The MCP3208\(^{598}\) is a 12-bit analog to digital converter with 8 channels (0-7).

- **channel**
  - The channel to read data from. The MCP3008/3208/3304 have 8 channels (0-7), while the MCP3004/3204/3302 have 4 channels (0-3), the MCP3002/3202 have 2 channels (0-1), and the MCP3001/3201/3301 only have 1 channel.

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

  Please refer to the device data-sheet to determine which channel is used as the relative base value (for example, when using an MCP3008 (page 151) 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).

16.2.9 MCP3301

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

The MCP3301\(^{599}\) 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).

16.2.10 MCP3302

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

The MCP3302\(^{600}\) 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 154) in differential mode, channel 0 is read relative to channel 1).

---

\(^{598}\) http://www.farnell.com/datasheets/808967.pdf

\(^{599}\) http://www.farnell.com/datasheets/1669397.pdf

\(^{600}\) http://www.farnell.com/datasheets/1486116.pdf
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).

16.2.11 MCP3304

class gpiozero.MCP3304(channel=0, differential=False, max_voltage=3.3, **spi_args)
The MCP3304 is a 12/13-bit analog to digital converter with 8 channels (0-7). When operated in differential mode, the device outputs a signed 13-bit value which is scaled from -1 to +1. When operated in single-ended mode (the default), the device outputs an unsigned 12-bit value scaled from 0 to 1.

channel
The channel to read data from. The MCP3008/3208/3304 have 8 channels (0-7), while the MCP3004/3204/3302 have 4 channels (0-3), 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 154) 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).

16.3 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 devices.

16.3.1 AnalogInputDevice

```python
class gpiozero.AnalogInputDevice(bits, max_voltage=3.3, **spi_args):
    Represents an analog input device connected to SPI (serial interface).

    Typical analog input devices are analog to digital converters\(^{602}\) (ADCs). Several classes are provided for specific ADC chips, including `MCP3004` (page 151), `MCP3008` (page 151), `MCP3204` (page 152), and `MCP3208` (page 153).

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

```from gpiozero import MCP3008

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

\(^{602}\) https://en.wikipedia.org/wiki/Analog-to-digital_converter
The \texttt{value} (page 156) 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 \texttt{PWMLED} (page 129) like so:

\begin{verbatim}
from gpiozero import MCP3008, PWMLED
pot = MCP3008(0)
led = PWMLED(17)
led.source = pot
\end{verbatim}

The \texttt{voltage} (page 156) attribute reports values between 0.0 and \texttt{max\_voltage} (which defaults to 3.3, the logic level of the GPIO pins).

\begin{description}
\item[bits] The bit-resolution of the device/channel.
\item[max\_voltage] The voltage required to set the device's value to 1.
\item[raw\_value] The raw value as read from the device.
\item[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).
\item[voltage] The current voltage read from the device. This will be a value between 0 and the \texttt{max\_voltage} parameter specified in the constructor.
\end{description}

16.3.2 SPI\text{Device}

class \texttt{gpiozero.SPIDevice(**spi\_args)}

Extends \texttt{Device} (page 203). Represents a device that communicates via the SPI protocol.

See \texttt{SPI keyword args} (page 149) for information on the keyword arguments that can be specified with the constructor.

\texttt{close()}  
Shut down the device and release all associated resources (such as GPIO pins).

This method is idempotent (can be called on an already closed device without any side-effects). It 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:

\begin{verbatim}
>>> from gpiozero import *
>>> bz = Buzzer(16)
>>> bz.on()
>>> bz.off()
>>> bz.close()
>>> led = LED(16)
>>> led.blink()
\end{verbatim}
Device (page 203) 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 156) method). Once a device is closed you can no longer use any other methods or properties to control or query the device.

---

603 https://docs.python.org/3.7/reference/compound_stmts.html#with
604 https://docs.python.org/3.7/library/constants.html#True
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 107) and API - Output Devices (page 127) 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.

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 LEDBoard

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

Extends LEDCollection (page 187) 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 159) instances to create trees of LEDs.
• **pwm** (bool) – If True, construct PWMLED (page 129) instances for each pin. If False (the default), construct regular LED (page 127) instances.

• **active_high** (bool) – If True (the default), the on() (page 161) method will set all the associated pins to HIGH. If False, the on() (page 161) method will set all pins to LOW (the off() (page 161) 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.

• **_order** (list or None) – If specified, this is the order of named items specified by keyword arguments (to ensure that the value tuple is constructed with a specific order). All keyword arguments must be included in the collection. If omitted, an alphabetically sorted order will be selected for keyword arguments.

• **pin_factory** (Factory or None) – See API - Pins (page 227) 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 159) 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).

---

605 https://docs.python.org/3.7/library/functions.html#bool
606 https://docs.python.org/3.7/library/constants.html#True
607 https://docs.python.org/3.7/library/constants.html#False
608 https://docs.python.org/3.7/library/functions.html#bool
609 https://docs.python.org/3.7/library/constants.html#True
610 https://docs.python.org/3.7/library/constants.html#False
611 https://docs.python.org/3.7/library/functions.html#bool
612 https://docs.python.org/3.7/library/constants.html#None
613 https://docs.python.org/3.7/library/constants.html#False
614 https://docs.python.org/3.7/library/constants.html#None
615 https://docs.python.org/3.7/library/constants.html#True
616 https://docs.python.org/3.7/library/stdtypes.html#list
617 https://docs.python.org/3.7/library/constants.html#None
618 https://docs.python.org/3.7/library/constants.html#None
619 https://docs.python.org/3.7/library/constants.html#None
620 https://docs.python.org/3.7/library/functions.html#float
621 https://docs.python.org/3.7/library/functions.html#float
622 https://docs.python.org/3.7/library/constants.html#False
623 https://docs.python.org/3.7/library/exceptions.html#ValueError
624 https://docs.python.org/3.7/library/functions.html#float
625 https://docs.python.org/3.7/library/constants.html#False
626 https://docs.python.org/3.7/library/exceptions.html#ValueError
• **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).

```python
off(*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 160) is currently active, it will be stopped first.

**Parameters**

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

```python
on(*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 160) is currently active, it will be stopped first.

**Parameters**

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

```python
pulse(fade_in_time=1, fade_out_time=1, 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\(^{638}\) or None\(^{639}\)) – Number of times to blink; None\(^{640}\) (the default) means forever.

- **background** (bool\(^{641}\)) – If True\(^{642}\) (the default), start a background thread to continue blinking and return immediately. If False\(^{643}\), 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 160) is currently active, it will be stopped first.

**Parameters** *args* (int\(^{644}\)) – The index(es) of the LED(s) to toggle. If no indexes are specified toggle the state of all LEDs.

### 17.1.2 LEDBarGraph

**class** `gpiozero.LEDBarGraph(*pins, pwm=False, active_high=True, initial_value=0.0, pin_factory=None)`

Extends LEDCollection (page 187) 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 163) and *values* (page 164) are supported:

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

(continues on next page)
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) — If True, construct PWMLED (page 129) instances for each pin. If False (the default), construct regular LED (page 127) instances. This parameter can only be specified as a keyword parameter.

- **active_high** (bool) — If True (the default), the on() method will set all the associated pins to HIGH. If False, the on() method will set all pins to LOW (the off() method always does the opposite). This parameter can only be specified as a keyword parameter.

- **initial_value** (float) — The initial value (page 163) 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 or None) — See API - Pins (page 227) 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 163), 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 163).

**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 163)

---

645 https://docs.python.org/3.7/library/functions.html#bool
646 https://docs.python.org/3.7/library/constants.html#True
647 https://docs.python.org/3.7/library/constants.html#False
648 https://docs.python.org/3.7/library/functions.html#bool
649 https://docs.python.org/3.7/library/constants.html#True
650 https://docs.python.org/3.7/library/constants.html#False
651 https://docs.python.org/3.7/library/functions.html#float
652 https://docs.python.org/3.7/library/constants.html#None
is effectively -1 < value <= 1.

values
An infinite iterator of values read from value (page 163).

17.1.3 LEDCharDisplay

class gpiozero.LEDCharDisplay(*pins, dp=None, font=None, pwm=False, active_high=True, initial_value=' ', pin_factory=None)

Extends LEDCollection (page 187) for a multi-segment LED display.

Multi-segment LED displays typically have 7 pins (labelled “a” through “g”) representing 7 LEDs laid out in a figure-of-8 fashion. Frequently, an eighth pin labelled “dp” is included for a trailing decimal-point:

```
  a
 f  b
 g
 e  c
   • dp
 d
```

Other common layouts are 9, 14, and 16 segment displays which include additional segments permitting more accurate renditions of alphanumerics. For example:

```
  a
 f  i  j  b
 k
 g  h
 e  n  c
 l  m
   • dp
 d
```

Such displays have either a common anode, or common cathode pin. This class defaults to the latter; when using a common anode display `active_high` should be set to `False`.

Instances of this class can be used to display characters or control individual LEDs on the display. For example:

```python
from gpiozero import LEDCharDisplay
char = LEDCharDisplay(4, 5, 6, 7, 8, 9, 10, active_high=False)
char.value = 'C'
```

If the class is constructed with 7 or 14 segments, a default `font` (page 165) will be loaded, mapping some ASCII characters to typical layouts. In other cases, the default mapping will simply assign “ ” (space) to all LEDs off. You can assign your own mapping at construction time or after instantiation.

While the example above shows the display with a `str` value, theoretically the `font` can map

---

653 https://en.wikipedia.org/wiki/Seven-segment_display
654 https://docs.python.org/3.7/library/constants.html#False
655 https://docs.python.org/3.7/library/stdtypes.html#str
any value that can be the key in a `dict`, so the value of the display can be likewise be any valid key value (e.g. you could map integer digits to LED patterns). That said, there is one exception to this: when `dp` is specified to enable the decimal-point, the `value` must be a `str` as the presence or absence of a “.” suffix indicates whether the `dp` LED is lit.

**Parameters**

- `*pins` – Specify the GPIO pins that the multi-segment display is attached to. Pins should be in the LED segment order A, B, C, D, E, F, G, and will be named automatically by the class. If a decimal-point pin is present, specify it separately as the `dp` parameter.
- `dp` (`int` or `str`) – If a decimal-point segment is present, specify it as this named parameter.
- `font` (`dict` or `None`) – A mapping of values (typically characters, but may also be numbers) to tuples of LED states. A default mapping for ASCII characters is provided for 7 and 14 segment displays.
- `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 opposite), the `on()` method will set all pins to LOW.
- `initial_value` – The initial value to display. Defaults to space (“”) which typically maps to all LEDs being inactive. If `None`, each device will be left in whatever state the pin is found in when configured for output (warning: this can be on).
- `pin_factory` (`Factory` or `None`) – See API - Pins (page 227) for more information (this is an advanced feature which most users can ignore).

**font**

An `LEDCharFont` mapping characters to tuples of LED states. The font is mutable after construction. You can assign a tuple of LED states to a character to modify the font, delete an existing character in the font, or assign a mapping of characters to tuples to replace the entire font.

Note that modifying the `font` never alters the underlying LED states. Only assignment to `value` (page 165), or calling the inherited `LEDCollection` methods (`on()`, `off()`, etc.) modifies LED states. However, modifying the font may alter the character returned by querying `value` (page 165).

**value**

The character the display should show. This is mapped by the current `font` to a tuple of LED states which is applied to the underlying LED objects when this attribute is set.

When queried, the current LED states are looked up in the font to determine the character shown. If the current LED states do not correspond to any character in the `font`,
the value is \texttt{None}\.\footnote{\url{https://docs.python.org/3.7/library/constants.html#None}}

It is possible for multiple characters in the font to map to the same LED states (e.g. S and 5). In this case, if the font was constructed from an ordered mapping (which is the default), then the first matching mapping will always be returned. This also implies that the value queried need not match the value set.

17.1.4 LEDMultiCharDisplay

class \texttt{gpiozero.LEDMultiCharDisplay}(char, *pins, active\_high=True, initial\_value=None, pin\_factory=None)

Wraps \texttt{LEDCharDisplay} (page 164) for multi-character multiplexed\footnote{\url{https://en.wikipedia.org/wiki/Multiplexed_display}} LED character displays.

The class is constructed with a \texttt{char} which is an instance of the \texttt{LEDCharDisplay} (page 164) class, capable of controlling the LEDs in one character of the display, and an additional set of \texttt{pins} that represent the common cathode (or anode) of each character.

\begin{quote}
\textbf{Warning:} You should not attempt to connect the common cathode (or anode) off each character directly to a GPIO. Rather, use a set of transistors (or some other suitable component capable of handling the current of all the segment LEDs simultaneously) to connect the common cathode to ground (or the common anode to the supply) and control those transistors from the GPIOs specified under \texttt{pins}.
\end{quote}

The \texttt{active\_high} parameter defaults to \texttt{True}\.\footnote{\url{https://docs.python.org/3.7/library/constants.html#True}} Note that it only applies to the specified \texttt{pins}, which are assumed to be controlling a set of transistors (hence the default). The specified \texttt{char} will use its own \texttt{active\_high} parameter. Finally, \texttt{initial\_value} defaults to a tuple of \texttt{value} (page 165) attribute of the specified display multiplied by the number of \texttt{pins} provided.

When the \texttt{value} (page 166) is set such that one or more characters in the display differ in value, a background thread is implicitly started to rotate the active character, relying on persistence of vision\footnote{\url{https://en.wikipedia.org/wiki/Persistence_of_vision}} to display the complete value.

\textbf{plex\_delay}

The delay (measured in seconds) in the loop used to switch each character in the multiplexed display on. Defaults to 0.005 seconds which is generally sufficient to provide a “stable” (non-flickery) display.

\textbf{value}

The sequence of values to display.

This can be any sequence containing keys from the \texttt{font} (page 165) of the associated character display. For example, if the value consists only of single-character strings, it’s valid to assign a string to this property (as a string is simply a sequence of individual character keys):

\begin{verbatim}
from gpiozero import LEDCharDisplay, LEDMultiCharDisplay

c = LEDCharDisplay(4, 5, 6, 7, 8, 9, 10)
d = LEDMultiCharDisplay(c, 19, 20, 21, 22)
d.value = 'LEDS'
\end{verbatim}

However, things get more complicated if a decimal point is in use as then this class needs to know explicitly where to break the value for use on each character of the display. This can be handled by simply assigning a sequence of strings thus:

from gpiozero import LEDCharDisplay, LEDMultiCharDisplay

c = LEDCharDisplay(4, 5, 6, 7, 8, 9, 10)
d = LEDMultiCharDisplay(c, 19, 20, 21, 22)
d.value = ('L', 'E', 'D', 'S')

This is how the value will always be represented when queried (as a tuple of individual values) as it neatly handles dealing with heterogeneous types and the aforementioned decimal point issue.

**Note:** The value also controls whether a background thread is in use to multiplex the display. When all positions in the value are equal the background thread is disabled and all characters are simultaneously enabled.

### 17.1.5 LEDCharFont

class gpiozero.LEDCharFont(font)
Contains a mapping of values to tuples of LED states.

This effectively acts as a “font” for LEDCharDisplay (page 164), and two default fonts (for 7-segment and 14-segment displays) are shipped with GPIO Zero by default. You can construct your own font instance from a dict which maps values (usually single-character strings) to a tuple of LED states:

```python
from gpiozero import LEDCharDisplay, LEDCharFont
my_font = LEDCharFont(
    {' ': (0, 0, 0, 0, 0, 0, 0),
     'D': (1, 1, 1, 1, 1, 1, 0),
     'A': (1, 1, 1, 0, 1, 1, 1),
     'd': (0, 1, 1, 1, 1, 0, 1),
     'a': (1, 1, 1, 1, 0, 1, 1),
    })
display = LEDCharDisplay(26, 13, 12, 22, 17, 19, 6, dp=5, font=my_font)
display.value = 'D'
```

Font instances are mutable and can be changed while actively in use by an instance of LEDCharDisplay (page 164). However, changing the font will not change the state of the LEDs in the display (though it may change the value (page 165) of the display when next queried).

**Note:** Your custom mapping should always include a value (typically space) which represents all the LEDs off. This will usually be the default value for an instance of LEDCharDisplay (page 164).

You may also wish to load fonts from a friendly text-based format. A simple parser for such formats (supporting an arbitrary number of segments) is provided by gpiozero.fonts.load_segment_font() (page 216).

### 17.1.6 ButtonBoard

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

Extends CompositeDevice (page 188) and represents a generic button board or collection of but-
tons. The value (page 169) 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 159) with a ButtonBoard (page 167):

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

leds = LEDBoard(2, 3, 4, 5)
bbtns = ButtonBoard(6, 7, 8, 9)
leds.source = btns

pause()
```

Alternatively you could represent the number of pressed buttons with an LEDBarGraph (page 162):

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

graph = LEDBarGraph(2, 3, 4, 5)
bb = ButtonBoard(6, 7, 8, 9)
graph.source = (mean(values) for values in bb.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 124) 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.

---

675 https://docs.python.org/3.7/library/functions.html#bool
676 https://docs.python.org/3.7/library/constants.html#None
677 https://docs.python.org/3.7/library/constants.html#True
678 https://docs.python.org/3.7/library/constants.html#False
679 https://docs.python.org/3.7/library/constants.html#None
680 https://docs.python.org/3.7/library/functions.html#bool
681 https://docs.python.org/3.7/library/constants.html#None
682 https://docs.python.org/3.7/library/functions.html#float
683 https://docs.python.org/3.7/library/constants.html#None
684 https://docs.python.org/3.7/library/functions.html#float
685 https://docs.python.org/3.7/library/functions.html#bool
686 https://docs.python.org/3.7/library/constants.html#True
If \texttt{False} (the default) the \texttt{when\_held} handler will be only be executed once per hold.

- \texttt{pin\_factory (Factory (page 232) or \texttt{None})} – See \textit{API - Pins} (page 227) for more information (this is an advanced feature which most users can ignore).
- \texttt{**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.

\begin{verbatim}
wait\_for\_press(timeout=None)
Pause the script until the device is activated, or the timeout is reached.

Parameters timeout (\texttt{float} or \texttt{None}) – Number of seconds to wait before proceeding. If this is \texttt{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 (\texttt{float} or \texttt{None}) – Number of seconds to wait before proceeding. If this is \texttt{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 \texttt{None}.

value
A \texttt{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 it will be passed as that parameter.

Set this property to \texttt{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 it will be passed as that parameter.
\end{verbatim}
Set this property to `None` (the default) to disable the event.

17.1.7 TrafficLights

```python
class gpiozero.TrafficLights(red=None, amber=None, green=None, *, pwm=False, initial_value=False, yellow=None, pin_factory=None):
    Extends LEDBoard (page 159) 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` or `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 129) instances to represent each LED. If `False` (the default), construct regular `LED` (page 127) 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 227) for more information (this is an advanced feature which most users can ignore).

### red

The red `LED` (page 127) or `PWMLED` (page 129).
amber
The amber LED (page 127) or PWMLED (page 129). Note that this attribute will not be present when the instance is constructed with the yellow keyword parameter.

yellow
The yellow LED (page 127) or PWMLED (page 129). Note that this attribute will only be present when the instance is constructed with the yellow keyword parameter.

green
The green LED (page 127) or PWMLED (page 129).

17.1.8 TrafficLightsBuzzer
class gpiozero.TrafficLightsBuzzer(lights, buzzer, button, *, pin_factory=None)
Extends CompositeOutputDevice (page 188) and is a generic class for HATs with traffic lights, a button and a buzzer.

Parameters
- **lights** (TrafficLights (page 170)) – An instance of TrafficLights (page 170) representing the traffic lights of the HAT.
- **buzzer** (Buzzer (page 134)) – An instance of Buzzer (page 134) representing the buzzer on the HAT.
- **button** (Button (page 107)) – An instance of Button (page 107) representing the button on the HAT.
- **pin_factory** (Factory (page 232) or None718) – See API - Pins (page 227) for more information (this is an advanced feature which most users can ignore).

lights
The TrafficLights (page 170) instance passed as the lights parameter.

buzzer
The Buzzer (page 134) instance passed as the buzzer parameter.

button
The Button (page 107) instance passed as the button parameter.

17.1.9 PiHutXmasTree
class gpiozero.PiHutXmasTree(*, pwm=False, initial_value=False, pin_factory=None)
Extends LEDBoard (page 159) for The Pi Hut’s Xmas board719: 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 172) attribute. Alternatively, as with all descendents of LEDBoard (page 159), you can treat the instance as a sequence of LEDs (the first element is the star (page 172)).

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()
```

(continues on next page)

718 https://docs.python.org/3.7/library/constants.html#None
719 https://thepihut.com/xmas
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`) – If `True`, construct `PWMLED` (page 129) instances for each pin. If `False` (the default), construct regular `LED` (page 127) 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 227) for more information (this is an advanced feature which most users can ignore).

`star`

Returns the `LED` (page 127) or `PWMLED` (page 129) representing the white star on top of the tree.

`led0, led1, led2, ...`

Returns the `LED` (page 127) or `PWMLED` (page 129) 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.

17.1.10 LedBorg

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

Extends `RGBLED` (page 131) for the PiBorg LedBorg: an add-on board containing a very bright RGB LED.

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

```python
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()
```
from gpiozero import LedBorg

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

Parameters

- **initial_value** (*Color* or *tuple*) - The initial color for the LedBorg. Defaults to black (0, 0, 0).
- **pwm** (*bool*) - If True (the default), construct `PWMLED` (page 129) instances for each component of the LedBorg. If False, construct regular `LED` (page 127) instances, which prevents smooth color graduations.
- **pin_factory** (*Factory* or *None*) - See *API - Pins* (page 227) for more information (this is an advanced feature which most users can ignore).

17.1.11 PiLiter

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

Extends `LEDBoard` (page 159) for the Ciseco Pi-LITEr: a strip of 8 very bright LEDs.

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

```python
from gpiozero import PiLiter

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

Parameters

- **pwm** (*bool*) - If True, construct `PWMLED` (page 129) instances for each pin. If False (the default), construct regular `LED` (page 127) 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 227) for more information (this is an advanced feature which most users can ignore).
17.1.12 PiLiterBarGraph

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

Extends LEDBarGraph (page 162) to treat the Ciseco Pi-LITEr\(^{746}\) 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`\(^{747}\)) – If True\(^{748}\), construct PWMLED (page 129) instances for each pin. If False\(^{749}\) (the default), construct regular LED (page 127) instances.

- **initial_value** (`float`\(^{750}\)) – The initial value of the graph given as a float between -1 and +1. Defaults to 0.0.

- **pin_factory** (`Factory` (page 232) or `None`\(^{751}\)) – See API - Pins (page 227) for more information (this is an advanced feature which most users can ignore).

17.1.13 PiTraffic

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

Extends TrafficLights (page 170) for the Low Voltage Labs PI-TRAFFIC\(^{752}\) 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:

```python
from gpiozero import PiTraffic

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

To use the PI-TRAFFIC board when attached to a non-standard set of pins, simply use the parent class, TrafficLights (page 170).

Parameters

- **pwm** (`bool`\(^{753}\)) – If True\(^{754}\), construct PWMLED (page 129) instances to represent each LED. If False\(^{755}\) (the default), construct regular LED (page 127) instances.

- **initial_value** (`bool`\(^{756}\)) – If False\(^{757}\) (the default), all LEDs will be off initially. If None\(^{758}\), each device will be left in whatever state the pin is found in.

---

\(^{746}\) [http://shop.ciseco.co.uk/pi-liter-8-led-strip-for-the-raspberry-pi/](http://shop.ciseco.co.uk/pi-liter-8-led-strip-for-the-raspberry-pi/)

\(^{747}\) [https://docs.python.org/3.7/library/functions.html#bool](https://docs.python.org/3.7/library/functions.html#bool)

\(^{748}\) [https://docs.python.org/3.7/library/constants.html#True](https://docs.python.org/3.7/library/constants.html#True)

\(^{749}\) [https://docs.python.org/3.7/library/constants.html#False](https://docs.python.org/3.7/library/constants.html#False)

\(^{750}\) [https://docs.python.org/3.7/library/functions.html#float](https://docs.python.org/3.7/library/functions.html#float)

\(^{751}\) [https://docs.python.org/3.7/library/constants.html#None](https://docs.python.org/3.7/library/constants.html#None)

\(^{752}\) [http://lowvoltagefabs.com/products/pi-traffic/](http://lowvoltagefabs.com/products/pi-traffic/)

\(^{753}\) [https://docs.python.org/3.7/library/functions.html#bool](https://docs.python.org/3.7/library/functions.html#bool)

\(^{754}\) [https://docs.python.org/3.7/library/constants.html#True](https://docs.python.org/3.7/library/constants.html#True)

\(^{755}\) [https://docs.python.org/3.7/library/constants.html#False](https://docs.python.org/3.7/library/constants.html#False)

\(^{756}\) [https://docs.python.org/3.7/library/functions.html#bool](https://docs.python.org/3.7/library/functions.html#bool)

\(^{757}\) [https://docs.python.org/3.7/library/constants.html#False](https://docs.python.org/3.7/library/constants.html#False)

\(^{758}\) [https://docs.python.org/3.7/library/constants.html#None](https://docs.python.org/3.7/library/constants.html#None)
when configured for output (warning: this can be on). If True, the device will be switched on initially.

- **pin_factory** *(Factory (page 232) or None)* – See API - Pins (page 227) for more information (this is an advanced feature which most users can ignore).

### 17.1.14 PiStop

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

Extends *TrafficLights* (page 170) 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+:

```python
from gpiozero import PiStop

traffic = PiStop('A+)
traffic.amber.on()
```

**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 129) instances to represent each LED. If False (the default), construct regular LED (page 127) 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 232) or None)* – See API - Pins (page 227) for more information (this is an advanced feature which most users can ignore).

### 17.1.15 FishDish

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

Extends *CompositeOutputDevice* (page 188) 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()
```

(continues on next page)
fish.button.wait_for_press()
fish.lights.on()

Parameters
- **pwm**: If True, construct `PWMLED` instances to represent each LED. If False (the default), construct regular `LED` instances.
- **pin_factory**: See API - Pins for more information (this is an advanced feature which most users can ignore).

17.1.16 TrafficHat

```python
from gpiozero import TrafficHat
hat = TrafficHat()
hat.button.wait_for_press()
hat.lights.on()
```

Parameters
- **pwm**: If True, construct `PWMLED` instances to represent each LED. If False (the default), construct regular `LED` instances.
- **pin_factory**: See API - Pins for more information (this is an advanced feature which most users can ignore).

17.1.17 TrafficpHat

```python
from gpiozero import TrafficpHat
phat = TrafficpHat()
phat.red.on()
phat.blink()
```

---

773 https://docs.python.org/3.7/library/functions.html#bool
774 https://docs.python.org/3.7/library/constants.html#True
775 https://docs.python.org/3.7/library/constants.html#False
776 https://docs.python.org/3.7/library/constants.html#None
777 https://uk.pi-supply.com/products/traffic-hat-for-raspberry-pi
778 https://docs.python.org/3.7/library/functions.html#bool
779 https://docs.python.org/3.7/library/constants.html#True
780 https://docs.python.org/3.7/library/constants.html#False
781 https://docs.python.org/3.7/library/constants.html#None
782 http://pisupp.ly/trafficphat
Parameters

- **pwm (bool) -** If `True`, construct `PWMLED` (page 129) instances to represent each LED. If `False` (the default), construct regular `LED` (page 127) 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 (page 232) or None) -** See `API - Pins` (page 227) for more information (this is an advanced feature which most users can ignore).

17.1.18 JamHat

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

Extends `CompositeOutputDevice` (page 188) 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` (page 129) instances to represent each LED on the board. If `False` (the default), construct regular `LED` (page 127) instances.
- **pin_factory (Factory (page 232) or None) -** See `API - Pins` (page 227) for more information (this is an advanced feature which most users can ignore).

`lights_1, lights_2`

Two `LEDBoard` (page 159) instances representing the top (`lights_1`) and bottom (`lights_2`) rows of LEDs on the JamHat.

`red, yellow, green`

`LED` (page 127) or `PWMLED` (page 129) 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 107) objects on the JamHat.

---

783 https://docs.python.org/3.7/library/functions.html#bool
784 https://docs.python.org/3.7/library/constantes.html#True
785 https://docs.python.org/3.7/library/constantes.html#False
786 https://docs.python.org/3.7/library/functions.html#bool
787 https://docs.python.org/3.7/library/constantes.html#None
788 https://docs.python.org/3.7/library/constantes.html#None
789 https://docs.python.org/3.7/library/constantes.html#None
790 https://docs.python.org/3.7/library/constantes.html#True
791 https://docs.python.org/3.7/library/constantes.html#None
792 https://thepihut.com/products/jam-hat
793 https://docs.python.org/3.7/library/constantes.html#bool
794 https://docs.python.org/3.7/library/constantes.html#True
795 https://docs.python.org/3.7/library/constantes.html#False
796 https://docs.python.org/3.7/library/constantes.html#None
buzzer

The `TonalBuzzer` (page 135) 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.

### 17.1.19 Pibrella

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

Extends `CompositeOutputDevice` (page 188) for the Cyntech/Pimoroni Pibrella board.

The Pibrella board comprises 3 LEDs, a button, a tonal buzzer, four general purpose input channels, and four general purpose output channels (with LEDs).

This class exposes the LEDs, button and buzzer.

Usage:

```python
from gpiozero import Pibrella

pb = Pibrella()

pb.button.wait_for_press()
pb.lights.on()
bpbuzzer.play('A4')

pb.off()
```

The four input and output channels are exposed so you can create GPIO Zero devices using these pins without looking up their respective pin numbers:

```python
from gpiozero import Pibrella, LED, Button

pb = Pibrella()
btn = Button(pb.inputs.a, pull_up=False)
led = LED(pb.outputs.e)

btn.when_pressed = led.on
```

Parameters

- `pwm` ([`bool`][798]) – If `True`,[799] construct `PWMLED` (page 129) instances to represent each LED on the board, otherwise if `False` (the default), construct regular `LED` (page 127) instances.

- `pin_factory` ([`Factory`][800] or `None`[801]) – See API - Pins (page 227) for more information (this is an advanced feature which most users can ignore).

`lights`

`TrafficLights` (page 170) instance representing the three LEDs

`red`, `amber`, `green`  

`LED` (page 127) or `PWMLED` (page 129) instances representing the red, amber, and green LEDs

---

[797] http://www.pibrella.com/
[798] https://docs.python.org/3.7/library/functions.html#bool
[799] https://docs.python.org/3.7/library/constants.html#True
[800] https://docs.python.org/3.7/library/constants.html#False
[801] https://docs.python.org/3.7/library/constants.html#None
button
The red Button (page 107) object on the Pibrella

buzzer
A TonalBuzzer (page 135) object representing the buzzer

inputs
A namedtuple()\(^{802}\) of the input pin numbers
a, b, c, d

outputs
A namedtuple()\(^{803}\) of the output pin numbers
e, f, g, h

off()
Turns all the LEDs off and stops the buzzer.

on()
Turns all the LEDs on and makes the buzzer play its mid tone.

17.1.20 Robot
class gpiozero.Robot(left, right, *, pin_factory=None)
Extends CompositeDevice (page 188) to represent a generic dual-motor robot.

This class is constructed with two motor instances representing the left and right wheels of the robot 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:

```python
from gpiozero import Robot
robot = Robot(left=Motor(4, 14), right=Motor(17, 18))
robot.forward()
```

Parameters
- **left** (Motor (page 137) or PhaseEnableMotor (page 138)) – A Motor (page 137) or a PhaseEnableMotor (page 138) for the left wheel of the robot.
- **right** (Motor (page 137) or PhaseEnableMotor (page 138)) – A Motor (page 137) or a PhaseEnableMotor (page 138) for the right wheel of the robot.
- **pin_factory** (Factory (page 232) or None\(^{804}\)) – See API - Pins (page 227) for more information (this is an advanced feature which most users can ignore).

left_motor
The Motor (page 137) on the left of the robot.

right_motor
The Motor (page 137) on the right of the robot.

backward(speed=1, *, curve_left=0, curve_right=0)
Drive the robot backward by running both motors backward.

Parameters
- **speed** (float\(^{805}\)) – Speed at which to drive the motors, as a value between 0 (stopped) and 1 (full speed). The default is 1.

---

\(^{802}\) https://docs.python.org/3.7/library/collections.html#collections.namedtuple
\(^{803}\) https://docs.python.org/3.7/library/collections.html#collections.namedtuple
\(^{804}\) https://docs.python.org/3.7/library/constants.html#None
\(^{805}\) https://docs.python.org/3.7/library/functions.html#float
• `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`.

```python
forward(speed=1, *, curve_left=0, curve_right=0)
```

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`.

```python
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.

```python
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.

```python
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.

```python
stop()
```

Stop the robot.

```python
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.

---

806. [https://docs.python.org/3.7/library/functions.html#float](https://docs.python.org/3.7/library/functions.html#float)
807. [https://docs.python.org/3.7/library/functions.html#float](https://docs.python.org/3.7/library/functions.html#float)
808. [https://docs.python.org/3.7/library/functions.html#float](https://docs.python.org/3.7/library/functions.html#float)
809. [https://docs.python.org/3.7/library/functions.html#float](https://docs.python.org/3.7/library/functions.html#float)
810. [https://docs.python.org/3.7/library/functions.html#float](https://docs.python.org/3.7/library/functions.html#float)
811. [https://docs.python.org/3.7/library/functions.html#float](https://docs.python.org/3.7/library/functions.html#float)
812. [https://docs.python.org/3.7/library/functions.html#float](https://docs.python.org/3.7/library/functions.html#float)
17.1.21 PhaseEnableRobot

17.1.22 RyanteckRobot

class gpiozero.RyanteckRobot(*, pwm=True, pin_factory=None)

Extends Robot (page 179) for the Ryanteck motor controller board\(^{813}\).

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] \(^{814}\)) – If True\(^{815}\) (the default), construct PWMOutputDevice (page 144) instances for the motor controller pins, allowing both direction and variable speed control. If False\(^{816}\), construct DigitalOutputDevice (page 143) instances, allowing only direction control.

- **pin_factory** ([Factory] (page 232) or None\(^{817}\)) – See API - Pins (page 227) for more information (this is an advanced feature which most users can ignore).

17.1.23 CamJamKitRobot

class gpiozero.CamJamKitRobot(*, pwm=True, pin_factory=None)

Extends Robot (page 179) for the CamJam #3 EduKit\(^{818}\) 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] \(^{819}\)) – If True\(^{820}\) (the default), construct PWMOutputDevice (page 144) instances for the motor controller pins, allowing both direction and variable speed control. If False\(^{821}\), construct DigitalOutputDevice (page 143) instances, allowing only direction control.

- **pin_factory** ([Factory] (page 232) or None\(^{822}\)) – See API - Pins (page 227) for more information (this is an advanced feature which most users can ignore).

\(^{813}\) https://uk.pi-supply.com/products/ryanteck-rtk-000-001-motor-controller-board-kit-raspberry-pi

\(^{814}\) https://docs.python.org/3.7/library/functions.html#bool

\(^{815}\) https://docs.python.org/3.7/library/constants.html#True

\(^{816}\) https://docs.python.org/3.7/library/constants.html#False

\(^{817}\) https://docs.python.org/3.7/library/constants.html#None

\(^{818}\) http://camjam.me/?page_id=1035

\(^{819}\) https://docs.python.org/3.7/library/functions.html#bool

\(^{820}\) https://docs.python.org/3.7/library/constants.html#True

\(^{821}\) https://docs.python.org/3.7/library/constants.html#False

\(^{822}\) https://docs.python.org/3.7/library/constants.html#None
17.1.24 PololuDRV8835Robot

class gpiozero.PololuDRV8835Robot(*, pwm=True, pin_factory=None)
  Extends Robot (page 179) for the Pololu DRV8835 Dual Motor Driver Kit.\(^{823}\)

  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)** \(^{824}\) – If True (the default), construct PWMOutputDevice (page 144) instances for the motor controller’s enable pins, allowing both direction and variable speed control. If False \(^{825}\), construct DigitalOutputDevice (page 143) instances, allowing only direction control.

- **pin_factory (Factory (page 232) or None)** \(^{826}\) – See API - Pins (page 227) for more information (this is an advanced feature which most users can ignore).

17.1.25 Energenie

class gpiozero.Energenie(socket=None, *, initial_value=False, pin_factory=None)
  Extends Device (page 203) 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)** \(^{830}\) – Which socket this instance should control. This is an integer number between 1 and 4.

- **initial_value (bool or None)** \(^{831}\) – The initial state of the socket. As Energenie sockets provide no means of reading their state, you may provide an initial state for the socket, which will be set upon construction. This defaults to False \(^{833}\), which will switch the socket off. Specifying None \(^{834}\) will not set any initial state nor transmit any control signal to the device.

- **pin_factory (Factory (page 232) or None)** \(^{835}\) – See API - Pins (page 227) for more information (this is an advanced feature which most users can ignore).

\(^{823}\) [https://www.pololu.com/product/2753](https://www.pololu.com/product/2753)
\(^{824}\) [https://docs.python.org/3.7/library/functions.html#bool](https://docs.python.org/3.7/library/functions.html#bool)
\(^{825}\) [https://docs.python.org/3.7/library/constants.html#True](https://docs.python.org/3.7/library/constants.html#True)
\(^{826}\) [https://docs.python.org/3.7/library/constants.html#False](https://docs.python.org/3.7/library/constants.html#False)
\(^{827}\) [https://docs.python.org/3.7/library/constants.html#None](https://docs.python.org/3.7/library/constants.html#None)
\(^{828}\) [https://energenie4u.co.uk/index.php/catalogue/product/ENER002-2PI](https://energenie4u.co.uk/index.php/catalogue/product/ENER002-2PI)
\(^{830}\) [https://docs.python.org/3.7/library/functions.html#int](https://docs.python.org/3.7/library/functions.html#int)
\(^{831}\) [https://docs.python.org/3.7/library/functions.html#bool](https://docs.python.org/3.7/library/functions.html#bool)
\(^{832}\) [https://docs.python.org/3.7/library/constants.html#None](https://docs.python.org/3.7/library/constants.html#None)
\(^{833}\) [https://docs.python.org/3.7/library/constants.html#False](https://docs.python.org/3.7/library/constants.html#False)
\(^{834}\) [https://docs.python.org/3.7/library/constants.html#None](https://docs.python.org/3.7/library/constants.html#None)
\(^{835}\) [https://docs.python.org/3.7/library/constants.html#None](https://docs.python.org/3.7/library/constants.html#None)
off()  
Turns the socket off.

on()  
Turns the socket on.

socket  
Returns the socket number.

value  
Returns True if the socket is on and False if the socket is off. Setting this property changes the state of the socket. Returns None only when constructed with initial_value set to None and neither on() (page 183) nor off() (page 183) have been called since construction.

17.1.26 StatusZero

class gpiozero.StatusZero(*labels, pwm=False, initial_value=False, pin_factory=None)
Extends LEDBoard (page 159) 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 127) 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.
- pwm (bool) – If True, construct PWMLED (page 129) instances to represent each LED. If False (the default), construct regular LED (page 127) 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 232) or None) – See API - Pins (page 227) for more information (this is an advanced feature which most users can ignore).
This entry represents one of the three labelled attributes supported on the STATUS Zero board. It is an `LEDBoard` (page 159) which contains:

**red**

The `LED` (page 127) or `PWMLED` (page 129) representing the red LED next to the label.

**green**

The `LED` (page 127) or `PWMLED` (page 129) representing the green LED next to the label.

### 17.1.27 StatusBoard

```python
class gpiozero.StatusBoard(*labels, pwm=False, initial_value=False, pin_factory=None)
```

Extends `CompositeOutputDevice` (page 188) 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')
status.wifi.lights.green.on()
status.wifi.button.when_pressed = status.wifi.lights.toggle
```

Each designated label will contain a “lights” `LEDBoard` (page 159) containing two `LED` (page 127) objects named “red” and “green”, and a `Button` (page 107) 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.
- **`pwm`** (`bool`) – If `True`, construct `PWMLED` (page 129) instances to represent each LED. If `False` (the default), construct regular `LED` (page 127) 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 227) for more information (this is an advanced feature which most users can ignore).

This entry represents one of the five labelled attributes supported on the STATUS board. It is an `CompositeOutputDevice` (page 188) which contains:

**lights**

A `LEDBoard` (page 159) representing the lights next to the label. It contains:

---

850 https://thepihut.com/status
851 https://docs.python.org/3.7/library/stdtypes.html#str
852 https://docs.python.org/3.7/library/functions.html#bool
853 https://docs.python.org/3.7/library/constants.html#True
854 https://docs.python.org/3.7/library/constants.html#False
855 https://docs.python.org/3.7/library/functions.html#bool
856 https://docs.python.org/3.7/library/constants.html#None
857 https://docs.python.org/3.7/library/constants.html#None
858 https://docs.python.org/3.7/library/constants.html#True
859 https://docs.python.org/3.7/library/constants.html#None
red
The LED (page 127) or PWMLED (page 129) representing the red LED next to the label.

green
The LED (page 127) or PWMLED (page 129) representing the green LED next to the label.

button
A Button (page 107) representing the button next to the label.

17.1.28 SnowPi

class gpiozero.SnowPi(*, pwm=False, initial_value=False, pin_factory=None)
Extends LEDBoard (page 159) for the Ryanteck SnowPi board.

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

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

Parameters

- `pwm (bool)` – If True, construct PWMLED (page 129) instances to represent each LED. If False (the default), construct regular LED (page 127) 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 227) for more information (this is an advanced feature which most users can ignore).

arms
A LEDBoard (page 159) representing the arms of the snow man. It contains the following attributes:

   - `left, right` Two LEDBoard (page 159) objects representing the left and right arms of the snow-man. They contain:

     - `top, middle, bottom` The LED (page 127) or PWMLED (page 129) down the snow-man’s arms.

eyes
A LEDBoard (page 159) representing the eyes of the snow-man. It contains:

   - `left, right` The LED (page 127) or PWMLED (page 129) for the snow-man’s eyes.
nose

The LED (page 127) or PWMLED (page 129) for the snow-man’s nose.

17.1.29 PumpkinPi

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

Extends LEDBoard (page 159) for the ModMyPi PumpkinPi\[869\] 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) – If True\[870\], construct PWMLED (page 129) instances to represent each LED. If False\[871\] (the default), construct regular LED (page 127) instances
- **initial_value** (bool or None) – If False\[872\] (the default), all LEDs will be off initially. If None\[873\], each device will be left in whatever state the pin is found in when configured for output (warning: this can be on). If True\[874\], the device will be switched on initially.
- **pin_factory** (Factory or None) – See API - Pins (page 227) for more information (this is an advanced feature which most users can ignore).

sides

A LEDBoard (page 159) representing the LEDs around the edge of the pumpkin. It contains:

- **left**, **right**

  Two LEDBoard (page 159) instances representing the LEDs on the left and right sides of the pumpkin. They each contain:

  - **top, midtop, middle, midbottom, bottom**

    Each LED (page 127) or PWMLED (page 129) around the specified side of the pumpkin.

eyes

A LEDBoard (page 159) representing the eyes of the pumpkin. It contains:

- **left**, **right**

  The LED (page 127) or PWMLED (page 129) for each of the pumpkin’s eyes.

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:

---

\[869\] https://www.modmypi.com/halloween-pumpkin-programmable-kit
\[870\] https://docs.python.org/3.7/library/functions.html#bool
\[871\] https://docs.python.org/3.7/library/constans.html#True
\[872\] https://docs.python.org/3.7/library/constans.html#False
\[873\] https://docs.python.org/3.7/library/functions.html#bool
\[874\] https://docs.python.org/3.7/library/constans.html#None
\[875\] https://docs.python.org/3.7/library/constans.html#False
\[876\] https://docs.python.org/3.7/library/constans.html#False
\[877\] https://docs.python.org/3.7/library/constans.html#True
\[878\] https://docs.python.org/3.7/library/constans.html#None
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.

17.2.1 LEDCollection

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

Extends CompositeOutputDevice (page 188). Abstract base class for LEDBoard (page 159) and LEDBarGraph (page 162).

is_lit

Composite devices are considered “active” if any of their constituent devices have a “truthy” value.
leds
A flat tuple of all LEDs contained in this collection (and all sub-collections).

### 17.2.2 CompositeOutputDevice

```python
class gpiozero.CompositeOutputDevice(*args, **kwargs)
```

Extends `CompositeDevice` (page 188) with `on()` (page 188), `off()` (page 188), and `toggle()` (page 188) methods for controlling subordinate output devices. Also extends `value` (page 188) to be writeable.

**Parameters**

- `*args` (*Device* (page 203)) – The un-named devices that belong to the composite device. The `value` (page 203) attributes of these devices will be represented within the composite device’s tuple `value` (page 188) in the order specified here.
- `_order` (*list*[^879] or *None*[^880]) – If specified, this is the order of named items specified by keyword arguments (to ensure that the `value` (page 188) 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 232) or *None*[^881]) – See API - Pins (page 227) for more information (this is an advanced feature which most users can ignore).
- `**kwargs` (*Device* (page 203)) – 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 188) attributes will be accessible as named elements of the composite device’s tuple `value` (page 188).

- `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.

### 17.2.3 CompositeDevice

```python
class gpiozero.CompositeDevice(*args, _order=None, pin_factory=None, **kwargs)
```

Extends `Device` (page 203). Represents a device composed of multiple devices like simple HATs, H-bridge motor controllers, robots composed of multiple motors, etc.

The constructor accepts subordinate devices as positional or keyword arguments. Positional arguments form unnamed devices accessed 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]
```

[^879]: https://docs.python.org/3.7/library/stdtypes.html#list
[^880]: https://docs.python.org/3.7/library/constants.html#None
[^881]: https://docs.python.org/3.7/library/constants.html#None
(continued from previous page)

```python
>>> d[1]
<gpiozero.LED object on pin GPIO2, active_high=True, is_active=False>
>>> d[2]
<gpiozero.LED object on pin GPIO3, active_high=True, is_active=False>
>>> d[3]
<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 203)) – The un-named devices that belong to the composite device. The `value` (page 190) attributes of these devices will be represented within the composite device’s tuple `value` (page 190) in the order specified here.*

- **`_order`** *(list or None)* – If specified, this is the order of named items specified by keyword arguments (to ensure that the `value` (page 190) 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 232) or None)* – See API - Pins (page 227) for more information (this is an advanced feature which most users can ignore).

- **`**kwargs`** *(Device (page 203)) – 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 190) attributes will be accessible as named elements of the composite device’s tuple `value` (page 190).*

**close()**

Shut down the device and release all associated resources (such as GPIO pins).

This method is idempotent (can be called on an already closed device without any side-effects). It 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 203) descendents can also be used as context managers using the `with` statement. For example:

---

882 https://docs.python.org/3.7/library/stdtypes.html#list
883 https://docs.python.org/3.7/library/constants.html#None
884 https://docs.python.org/3.7/library/constants.html#None
885 https://docs.python.org/3.7/reference/compound_stmts.html#with

17.2. Base Classes 189
```
>>> from gpiozero import *
>>> with Buzzer(16) as bz:
...    bz.on()
...

>>> with LED(16) as led:
...    led.on()
...
```

**closed**

Returns `True`\(^{886}\) if the device is closed (see the `close()` (page 189) 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()`\(^{887}\) type constructed to represent the value of the composite device. The `value` (page 190) attribute returns values of this type.

**value**

A `namedtuple()`\(^{888}\) 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.

---

\(^{886}\) [https://docs.python.org/3.7/library//constants.html#True](https://docs.python.org/3.7/library/constants.html#True)

\(^{887}\) [https://docs.python.org/3.7/library/collections.html#collections.namedtuple](https://docs.python.org/3.7/library/collections.html#collections.namedtuple)

\(^{888}\) [https://docs.python.org/3.7/library/collections.html#collections.namedtuple](https://docs.python.org/3.7/library/collections.html#collections.namedtuple)
CHAPTER 18

API - Internal Devices

GPIO Zero also provides several “internal” devices which represent facilities provided by the operating system itself. These can be used to react to things like the time of day, or whether a server is available on the network.

These devices provide an API similar to and compatible with GPIO devices so that internal device events can trigger changes to GPIO output devices the way input devices can. In the same way a Button (page 107) object is active when it’s pressed, and can be used to trigger other devices when its state changes, a TimeOfDay (page 192) object is active during a particular time period.

Consider the following code in which a Button (page 107) object is used to control an LED (page 127) object:

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

led = LED(2)
btn = Button(3)

btn.when_pressed = led.on
btn.when_released = led.off

pause()
```

Now consider the following example in which a TimeOfDay (page 192) object is used to control an LED (page 127) using the same method:

```python
from gpiozero import LED, TimeOfDay
from datetime import time
from signal import pause

led = LED(2)
tod = TimeOfDay(time(9), time(10))

tod.when_activated = led.on
tod.when_deactivated = led.off

pause()
```
Here, rather than the LED being controlled by the press of a button, it’s controlled by the time. When the time reaches 09:00AM, the LED comes on, and at 10:00AM it goes off.

Like the Button (page 107) object, internal devices like the TimeOfDay (page 192) object has value (page 193), values, is_active (page 193), when_activated (page 193) and when_deactivated (page 193) attributes, so alternative methods using the other paradigms would also work.

Note: Note that although the constructor parameter pin_factory is available for internal devices, and is required to be valid, the pin factory chosen will not make any practical difference. Reading a remote Pi’s CPU temperature, for example, is not currently possible.

18.1 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).

18.1.1 TimeOfDay

class gpiozero.TimeOfDay(start_time, end_time, *, utc=True, event_delay=5.0, pin_factory=None)

Extends PolledInternalDevice (page 199) 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 182) plug between 07:00AM and 08:00AM:

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

lamp = Energenie(1)
morning = TimeOfDay(time(7), time(8))

morning.when_activated = lamp.on
morning.when_deactivated = lamp.off

pause()
```

Note that start_time may be greater than end_time, indicating a time period which crosses midnight.

Parameters

- **start_time** (time) – The time from which the device will be considered active.
- **end_time** (time) – The time after which the device will be considered inactive.
- **utc** (bool) – If True (the default), a naive UTC time will be used for the comparison rather than a local time-zone reading.
- **event_delay** (float) – The number of seconds between file reads (defaults

889 https://docs.python.org/3.7/library/datetime.html#datetime.time
890 https://docs.python.org/3.7/library/datetime.html#datetime.time
891 https://docs.python.org/3.7/library/datetime.html#datetime.time
892 https://docs.python.org/3.7/library/functions.html#bool
893 https://docs.python.org/3.7/library/const.pdf.html#True
894 https://docs.python.org/3.7/library/functions.html#float
to 10 seconds).

- **pin_factory** (*Factory* (page 232) or *None*) – See *API - Pins* (page 227) 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.

**is_active**
Returns *True* if the device is currently active and *False* otherwise. This property is usually derived from *value* (page 193). Unlike *value* (page 193), this is always a boolean.

**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 1 when the system clock reads between *start_time* (page 193) and *end_time* (page 193), and 0 otherwise. If *start_time* (page 193) is greater than *end_time* (page 193) (indicating a period that crosses midnight), then this returns 1 when the current time is greater than *start_time* (page 193) or less than *end_time* (page 193).

**when_activated**
The function to run when the device changes state from inactive to active (time reaches *start_time*).

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 it 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 (time reaches *end_time*).

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 it will be passed as that parameter.

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

### 18.1.2 PingServer

**class gpiozero.PingServer(host, *, event_delay=10.0, pin_factory=None)**

Extends *PollableInternalDevice* (page 199) to provide a device which is active when a *host* (domain name or IP address) can be pinged.

The following example lights an LED while *google.com* is reachable:

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

google = PingServer('google.com')
led = LED(4)
```

(continues on next page)
pass

Parameters

- **host** *(str)* – The hostname or IP address to attempt to ping.
- **event_delay** *(float)* – The number of seconds between pings (defaults to 10 seconds).
- **pin_factory** *(Factory (page 232) or None)* – See *API - Pins* (page 227) for more information (this is an advanced feature which most users can ignore).

**host**

The hostname or IP address to test whenever *value* (page 194) is queried.

**is_active**

Returns *True* if the device is currently active and *False* otherwise. This property is usually derived from *value* (page 194). Unlike *value* (page 194), this is *always* a boolean.

**value**

Returns 1 if the host returned a single ping, and 0 otherwise.

**when_activated**

The function to run when the device changes state from inactive (host unresponsive) to active (host responsive).

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 it 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 inactive (host responsive) to active (host unresponsive).

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 it will be passed as that parameter.

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

### 18.1.3 CPUTemperature

class gpiozero.CPUTemperature(sensor_file='/sys/class/thermal/thermal_zone0/temp', *, min_temp=0.0, max_temp=100.0, threshold=80.0, event_delay=5.0, pin_factory=None)

Extends *PolledInternalDevice* (page 199) 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:

---

https://docs.python.org/3.7/library/stdtypes.html#str
https://docs.python.org/3.7/library/functions.html#float
https://docs.python.org/3.7/library/constants.html#None
https://docs.python.org/3.7/library/constants.html#True
https://docs.python.org/3.7/library/constants.html#False
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: {:.0f}°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 milli-degrees celsius.

- **min_temp** (*float*) – The temperature at which `value` (page 195) will read 0.0. This defaults to 0.0.

- **max_temp** (*float*) – The temperature at which `value` (page 195) 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 195)). This defaults to 80.0.

- **event_delay** (*float*) – The number of seconds between file reads (defaults to 5 seconds).

- **pin_factory** (*Factory* (page 232) or *None*) – See API - Pins (page 227) for more information (this is an advanced feature which most users can ignore).

`is_active`

Returns `True` when the CPU `temperature` (page 195) exceeds the `threshold`.

`temperature`

Returns the current CPU temperature in degrees celsius.

`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 195) is `temperature` (page 195) divided by 100 by default.

`when_activated`

The function to run when the device changes state from inactive to active (temperature reaches `threshold`).

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 it will be passed as that parameter.

904 https://docs.python.org/3.7/library/stdtypes.html#str
905 https://docs.python.org/3.7/library/functions.html#float
906 https://docs.python.org/3.7/library/functions.html#float
907 https://docs.python.org/3.7/library/functions.html#float
908 https://docs.python.org/3.7/library/functions.html#float
909 https://docs.python.org/3.7/library/constants.html#None
910 https://docs.python.org/3.7/library/constants.html#True
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 (temperature drops below `threshold`).

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 it will be passed as that parameter.

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

### 18.1.4 LoadAverage

```python
class gpiozero.LoadAverage(load_average_file='/proc/loadavg', *, min_load_average=0.0, max_load_average=1.0, threshold=0.8, minutes=5, event_delay=10.0, pin_factory=None)
```

Extends `PolledInternalDevice` (page 199) 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 197) will read 0.0. This defaults to 0.0.

- `max_load_average` *(float) –* The load average at which `value` (page 197) 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 197)). 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.

- `event_delay` *(float) –* The number of seconds between file reads (defaults to 10 seconds).
• **pin_factory** *(Factory (page 232) or None)* – See *API - Pins* (page 227) for more information (this is an advanced feature which most users can ignore).

**is_active**

Returns `True` when the `load_average` (page 197) 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.

**when_activated**

The function to run when the device changes state from inactive to active (load average reaches `threshold`).

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 it 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 (load average drops below `threshold`).

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 it will be passed as that parameter.

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

### 18.1.5 DiskUsage

**class gpiozero.DiskUsage** *(filesystem='/', *, threshold=90.0, event_delay=30.0, pin_factory=None)*

Extends `PolledInternalDevice` (page 199) 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**

917 https://docs.python.org/3.7/library/constants.html#None  
918 https://docs.python.org/3.7/library/constants.html#True
- **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 198)). This defaults to 90.0.

- **event_delay** *(float)* – The number of seconds between file reads (defaults to 30 seconds).

- **pin_factory** *(Factory (page 232) or None)* – See **API - Pins** (page 227) for more information (this is an advanced feature which most users can ignore).

### is_active

Returns `True` when the disk `usage` (page 198) 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 198) by 100.

### when_activated

The function to run when the device changes state from inactive to active (disk usage reaches `threshold`).

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 it 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 (disk usage drops below `threshold`).

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 it will be passed as that parameter.

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

## 18.2 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):

---

[919] https://docs.python.org/3.7/library/stdtypes.html#str
[920] https://docs.python.org/3.7/library/functions.html#float
[921] https://docs.python.org/3.7/library/functions.html#float
[922] https://docs.python.org/3.7/library/constants.html#None
[923] https://docs.python.org/3.7/library/constants.html#True
The following sections document these base classes for advanced users that wish to construct classes for their own devices.

### 18.2.1 PolledInternalDevice

```python
class gpiozero.PolledInternalDevice(*, event_delay=1.0, pin_factory=None)
```

Extends `InternalDevice` (page 199) to provide a background thread to poll internal devices that lack any other mechanism to inform the instance of changes.

### 18.2.2 InternalDevice

```python
class gpiozero.InternalDevice(*, pin_factory=None)
```

Extends `Device` (page 203) 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 203) is the root of the hierarchy, implementing base functionality like `close()` (page 203) and context manager handlers.
- **GPIODevice** (page 125) represents individual devices that attach to a single GPIO pin
- **SPIDevice** (page 156) represents devices that communicate over an SPI interface (implemented as four GPIO pins)
- **InternalDevice** (page 199) represents devices that are entirely internal to the Pi (usually operating system related services)
- **CompositeDevice** (page 188) represents devices composed of multiple other devices like HATs

There are also several mixin classes\textsuperscript{924} 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):

\textsuperscript{924} [https://en.wikipedia.org/wiki/Mixin](https://en.wikipedia.org/wiki/Mixin)
19.1 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 203) property, the value (page 203) property, and the close() (page 203)
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 (such as GPIO pins).

This method is idempotent (can be called on an already closed device without any side-effects).
It 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 203) 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 203) 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 203). Unlike value (page 203), this is always a boolean.

925 https://docs.python.org/3.7/reference/compound_stmts.html#with
926 https://docs.python.org/3.7/library/constants.html#True
927 https://docs.python.org/3.7/library/constants.html#False
928 https://docs.python.org/3.7/library/constants.html#False
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.

19.2 ValuesMixin

class gpiozero.ValuesMixin(...)
Adds a values (page 204) property to the class which returns an infinite generator of readings from the value (page 203) property. There is rarely a need to use this mixin directly as all base classes in GPIO Zero include it.

Note: Use this mixin first in the parent class list.

values
An infinite iterator of values read from value.

19.3 SourceMixin

class gpiozero.SourceMixin(...)
Adds a source (page 204) property to the class which, given an iterable or a ValuesMixin (page 204) descendent, sets value (page 203) to each member of that iterable until it is exhausted. This mixin is generally included in novel output devices to allow their state to be driven from another device.

Note: Use this mixin first in the parent class list.

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

source_delay
The delay (measured in seconds) in the loop used to read values from source (page 204). Defaults to 0.01 seconds which is generally sufficient to keep CPU usage to a minimum while providing adequate responsiveness.

19.4 SharedMixin

class gpiozero.SharedMixin(...)
This mixin marks a class as “shared”. In this case, the meta-class (GPIOMeta) will use _shared_key() (page 204) to convert the constructor arguments to an immutable key, and will check whether any existing instances match that key. If they do, they will be returned by the constructor instead of a new instance. An internal reference counter is used to determine how many times an instance has been “constructed” in this way.

When close() (page 203) is called, an internal reference counter will be decremented and the instance will only close when it reaches zero.

classmethod _shared_key(*args, **kwargs)
This is called with the constructor arguments to generate a unique key (which must be storable in a dict and, thus, immutable and hashable) representing the instance that can be shared. This must be overridden by descendents.

929 https://docs.python.org/3.7/library/stdtypes.html#dict
19.5 EventsMixin

class gpiozero.EventsMixin(...)

Advises event-detection when_activated() (page 205) and when_deactivated() (page 205) events to a device based on changes to the is_active (page 203) property common to all devices. Also adds wait_for_active() (page 205) and wait_for_inactive() (page 205) 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\(^{930}\) or None\(^{931}\)) – Number of seconds to wait before proceeding. If this is None\(^{932}\) (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\(^{933}\) or None\(^{934}\)) – Number of seconds to wait before proceeding. If this is None\(^{935}\) (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\(^{936}\).

inactive_time

The length of time (in seconds) that the device has been inactive for. When the device is active, this is None\(^{937}\).

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 it will be passed as that parameter.

Set this property to None\(^{938}\) (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 it will be passed as that parameter.

Set this property to None\(^{939}\) (the default) to disable the event.

930 https://docs.python.org/3.7/library/functions.html#float
931 https://docs.python.org/3.7/library/constants.html#None
932 https://docs.python.org/3.7/library/constants.html#None
933 https://docs.python.org/3.7/library/functions.html#float
934 https://docs.python.org/3.7/library/constants.html#None
935 https://docs.python.org/3.7/library/constants.html#None
936 https://docs.python.org/3.7/library/constants.html#None
937 https://docs.python.org/3.7/library/constants.html#None
938 https://docs.python.org/3.7/library/constants.html#None
939 https://docs.python.org/3.7/library/constants.html#None
19.6 HoldMixin

```python
class gpiozero.HoldMixin(...)
    Extends EventsMixin (page 205) to add the when_held (page 206) event and the machinery to fire that event repeatedly (when hold_repeat (page 206) is True) at internals defined by hold_time (page 206).

    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 206) event rather than when the device activated, in contrast to active_time (page 205). If the device is not currently held, this is None.

    hold_repeat
        If True, when_held (page 206) will be executed repeatedly with hold_time (page 206) 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 206) handler. If hold_repeat (page 206) is True, this is also the length of time between invocations of when_held (page 206).

    is_held
        When True, the device has been active for at least hold_time (page 206) seconds.

    when_held
        The function to run when the device has remained active for hold_time (page 206) seconds. This can be set to a function which accepts no (mandatory) parameters, or a Python function which accepts a single mandatory parameter (with as many optional parameters as you like). If the function accepts a single mandatory parameter, the device that activated will be passed as that parameter.

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

---

940 https://docs.python.org/3.7/library/constants.html#True
941 https://docs.python.org/3.7/library/constants.html#None
942 https://docs.python.org/3.7/library/constants.html#True
943 https://docs.python.org/3.7/library/constants.html#True
944 https://docs.python.org/3.7/library/constants.html#None
API - Device Source Tools

GPIO Zero includes several utility routines which are intended to be used with the *Source/Values* (page 65) 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* (page 204) and *values* (page 204) deal with infinite iterators, another excellent source of utilities is the *itertools*945 module in the standard library.

### 20.1 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 hysteresis946 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:

---

945 [https://docs.python.org/3.7/library/itertools.html#module-itertools](https://docs.python.org/3.7/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:

```python
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:

```python
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:

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

led = LED(4)
```

---

947 https://docs.python.org/3.7/library/constants.html#True
948 https://docs.python.org/3.7/library/constants.html#False
949 https://docs.python.org/3.7/library/constants.html#False
950 https://docs.python.org/3.7/library/constants.html#True
btn = Button(17)
led.source = negated(btn)
pause()

gpiozero.tools.post_delayed(values, delay)
Waits for delay seconds after returning each item from values.

gpiozero.tools.post_periodic_filtered(values, repeat_after, block)
After every repeat_after items, blocks the next block items from values. Note that unlike pre_periodic_filtered() (page 209), 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()
```
20.2 Combining sources

`gpiozero.tools.all_values(*values)`
Returns the logical conjunction\(^{951}\) of all supplied values (the result is only `True`\(^{952}\) if and only if all input values are simultaneously `True`\(^{953}\)). One or more `values` can be specified. For example, to light an LED (page 127) 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\(^{954}\) of all supplied values (the result is `True`\(^{955}\) if any of the input values are currently `True`\(^{956}\)). One or more `values` can be specified. For example, to light an LED (page 127) 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 129) as the average of several potentiometers connected to an MCP3008 (page 151) ADC:

```python
from gpiozero import MCP3008, PWMLED
from gpiozero.tools import averaged
from signal import pause

pot1 = MCP3008(channel=0)
```

\(^{951}\) [https://en.wikipedia.org/wiki/Logical_conjunction](https://en.wikipedia.org/wiki/Logical_conjunction)

\(^{952}\) [https://docs.python.org/3.7/library/constants.html#True](https://docs.python.org/3.7/library/constants.html#True)

\(^{953}\) [https://docs.python.org/3.7/library/constants.html#True](https://docs.python.org/3.7/library/constants.html#True)

\(^{954}\) [https://en.wikipedia.org/wiki/Logical_disjunction](https://en.wikipedia.org/wiki/Logical_disjunction)

\(^{955}\) [https://docs.python.org/3.7/library/constants.html#True](https://docs.python.org/3.7/library/constants.html#True)

\(^{956}\) [https://docs.python.org/3.7/library/constants.html#True](https://docs.python.org/3.7/library/constants.html#True)
```python
#gpiozero 1.6.1 Documentation

(continued from previous page)

```python
pot2 = MCP3008(channel=1)
pot3 = MCP3008(channel=2)
led = PWMLED(4)

led.source = averaged(pot1, pot2, pot3)
pause()
```

```python
# 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 129) as the product (i.e. multiplication) of several potentiometers connected to an `MCP3008` (page 151) 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()
```

```python
# gpiozero.tools.summed(*values)

Returns the sum of all supplied values. One or more `values` can be specified. For example, to light a `PWMLED` (page 129) as the (scaled) sum of several potentiometers connected to an `MCP3008` (page 151) 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()
```

```python
# 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)
```

(continues on next page)
zip_values(left, right) is equivalent to zip(left.values, right.values).

## 20.3 Artificial sources

**gpiozero.tools.alternating_values(initial_value=False)**  
Provides an infinite source of values alternating between `True` and `False`, starting with `initial_value` (which defaults to `False`). For example, to produce a flashing LED:

```python
from gpiozero import LED
from gpiozero.tools import alternating_values
from signal import pause

red = LED(2)
red.source_delay = 0.5
red.source = alternating_values()
pause()
```

**gpiozero.tools.cos_values(period=360)**  
Provides an infinite source of values representing a cosine wave (from -1 to +1) which repeats every `period` values. For example, to produce a “siren” effect with a couple of LEDs that repeats once a second:

```python
from gpiozero import PWMLED
from gpiozero.tools import cos_values, scaled_half, 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_half(cos_values(100))
blue.source = inverted(red)
pause()
```

If you require a different range than -1 to +1, see `scaled()` (page 210).

**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.source_delay = 0.01
red.source = ramping_values(100)

pause()

If you require a wider range than 0 to 1, see scaled() (page 210).

**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 210).

**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_half, 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_half(sin_values(100))
blue.source = inverted(red)
pause()
```

If you require a different range than -1 to +1, see scaled() (page 210).
GPIO Zero includes a concept of “fonts” which is somewhat different to that you may be familiar with. While a typical printing font determines how a particular character is rendered on a page, a GPIO Zero font determines how a particular character is rendered by a series of lights, like LED segments (e.g. with LEDCharDisplay (page 164) or LEDMultiCharDisplay (page 166)).

As a result, GPIO Zero’s fonts are quite crude affairs, being little more than mappings of characters to tuples of LED states. Still, it helps to have a “friendly” format for creating such fonts, and in this module the library provides several routines for this purpose.

The module itself is typically imported as follows:

```python
from gpiozero import fonts
```

### 21.1 Font Parsing

`gpiozero.fonts.load_font_7seg(filename_or_obj)`

Given a filename or a file-like object, parse it as an font definition for a 7-segment display\(^{560}\), returning a `dict`\(^{561}\) suitable for use with LEDCharDisplay (page 164).

The file-format is a simple text-based format in which blank and #-prefixed lines are ignored. All other lines are assumed to be groups of character definitions which are cells of 3x3 characters laid out as follows:

```
Ca
fgb
edc
```

Where C is the character being defined, and a-g define the states of the LEDs for that position. a, d, and g are on if they are “_”. b, c, e, and f are on if they are “|”. Any other character in these positions is considered off. For example, you might define the following characters:

```
  . 0_ 1. 2_ 3_ 4. 5_ 6_ 7_ 8_ 9_
  ... |l| ..| ..| ..| ..| ..| ..| ..| ..| ..|  
  ... |l| ..| ..| ..| ..| ..| ..| ..| ..| ..|  
```

\(^{560}\) [https://en.wikipedia.org/wiki/Seven-segment_display](https://en.wikipedia.org/wiki/Seven-segment_display)

\(^{561}\) [https://docs.python.org/3.7/library/stdtypes.html#dict](https://docs.python.org/3.7/library/stdtypes.html#dict)
In the example above, empty locations are marked with “.” but could mostly be left as spaces. However, the first item defines the space (“ ”) character and needs some non-space characters in its definition as the parser also strips empty columns (as typically occur between character definitions). This is also why the definition for “1” must include something to fill the middle column.

In the example above, several locations have extraneous characters. For example, the “/” in the center of the “0” definition, or the “-” in the middle of the “8”. These locations are ignored, but filled in nonetheless to make the shape more obvious.

These extraneous locations could equally well be left as spaces. However, the first item defines the space (“ ”) character and needs some non-space characters in its definition as the parser also strips empty columns (as typically occur between character definitions) and verifies that definitions are 5 columns wide and 5 rows high.

This also explains why place-holder characters (“.”) have been inserted at the top of the definition of the “1” character. Otherwise the parser will strip these empty columns and decide the definition is invalid (as the result is only 3 columns wide).

### gpiozero.fonts.load_segment_font(filename_or_obj, width, height, pins)

A generic function for parsing segment font definition files.

If you’re working with “standard” 7-segment964 or 14-segment965 displays you don’t want this function; see load_font_7seg() (page 215) or load_font_14seg() (page 216) instead. However, if you are working with another style of segmented display and wish to construct a parser for a custom format, this is the function you want.

The filename_or_obj parameter is simply the file-like object or filename to load. This is typically passed in from the calling function.

The width and height parameters give the width and height in characters of each character definition. For example, these are 3 and 3 for 7-segment displays. Finally, pins is a list of tuples that defines the position of each pin definition in the character array, and the character that marks that position “active”.

---

962 https://en.wikipedia.org/wiki/Fourteen-segment_display
963 https://docs.python.org/3.7/library/stdtypes.html#dict
964 https://en.wikipedia.org/wiki/Seven-segment_display
965 https://en.wikipedia.org/wiki/Fourteen-segment_display
For example, for 7-segment displays this function is called as follows:

```python
load_segment_font(filename_or_obj, width=3, height=3, pins=[
    (1, '_'), (5, '|'), (8, '|'), (7, '_'),
    (6, '|'), (3, '|'), (4, '_')])
```

This dictates that each character will be defined by a 3x3 character grid which will be converted into a nine-character string like so:

<table>
<thead>
<tr>
<th>012</th>
<th>345</th>
<th>678</th>
</tr>
</thead>
<tbody>
<tr>
<td>'012345678'</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Position 0 is always assumed to be the character being defined. The `pins` list then specifies: the first pin is the character at position 1 which will be “on” when that character is “_”. The second pin is the character at position 5 which will be “on” when that character is “|”, and so on.
GPIO Zero includes a *Tone* (page 219) class intended for use with the *TonalBuzzer* (page 135). This class is in the `tones` module of GPIO Zero and is typically imported as follows:

```python
from gpiozero.tones import Tone
```

### 22.1 Tone

class `gpiozero.tones.Tone`

Represents a frequency of sound in a variety of musical notations. *Tone* (page 219) class can be used with the *TonalBuzzer* (page 135) class to easily represent musical tones. The class can be constructed in a variety of ways. For example as a straight frequency in Hz\(^6\) (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 \(A\)^7 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 220), `from_midi()` (page 220), or `from_note()` (page 220), 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)
```

:\(^6\) https://en.wikipedia.org/wiki/Hertz
\(^7\) https://en.wikipedia.org/wiki/Concert_pitch
Several attributes are provided to permit conversion to any of the supported construction formats: `frequency` (page 220), `midi` (page 220), and `note` (page 220). Methods are provided to step `up()` (page 220) or `down()` (page 220) to adjacent MIDI notes.

### Warning
Currently `Tone` (page 219) derives from `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.

```python
>>> Tone(midi=69)
>>> Tone(note='A4')
```

**down** *(n=1)*
Return the `Tone` (page 219) n semi-tones below this frequency (n defaults to 1).

**classmethod from_frequency** *(freq)*
Construct a `Tone` (page 219) from a frequency specified in Hz which must be a positive floating-point value in the range 0 < freq <= 20000.

**classmethod from_midi** *(midi_note)*
Construct a `Tone` (page 219) 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.

**classmethod from_note** *(note)*
Construct a `Tone` (page 219) 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.

```python
>>> up(n=1)
```

Return the `Tone` (page 219) n semi-tones above this frequency (n defaults to 1).

**frequency**
Return the frequency of the tone in Hz.

**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) `ValueError` exception will be raised.

**note**
Return the (nearest) note to the tone’s frequency. This will be a string in the form accepted by `from_note()` (page 220). If the frequency is outside the range represented by this format (“A0” is approximately 27.5Hz, and “G9” is approximately 12.5Khz) a `ValueError` exception will be raised.

---

968 https://docs.python.org/3.7/library/functions.html#float
969 https://en.wikipedia.org/wiki/Hertz
970 https://en.wikipedia.org/wiki/Concert_pitch
971 https://en.wikipedia.org/wiki/Concert_pitch
972 https://en.wikipedia.org/wiki/Hertz
973 https://docs.python.org/3.7/library/exceptions.html#ValueError
974 https://docs.python.org/3.7/library/exceptions.html#ValueError
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:

### 23.1 pi_info

```python
gpiozero.pi_info(revision=None)
```

Returns a `PiBoardInfo` (page 221) instance containing information about a revision of the Raspberry Pi.

**Parameters**
- `revision (str)` – The revision of the Pi to return information about. If this is omitted or `None` (the default), then the library will attempt to determine the model of Pi it is running on and return information about that.

### 23.2 PiBoardInfo

```python
class gpiozero.PiBoardInfo
```

This class is a `namedtuple` derivative used to represent information about a particular model of Raspberry Pi. While it is a tuple, it is strongly recommended that you use the following named attributes to access the data contained within. The object can be used in format strings with various custom format specifications:

```python
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()))
```

---

975 https://docs.python.org/3.7/library/stdtypes.html#str
976 https://docs.python.org/3.7/library/constants.html#None
977 https://docs.python.org/3.7/library/collections.html#collections.namedtuple
“color” and “mono” can be prefixed to format specifications to force the use of ANSI color codes\(^\text{978}\). If 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 251). If multiple pins support the desired function, `PinMultiplePins` (page 251) will be raised (use `physical_pins()` (page 222) 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\(^\text{980}\) which is not indexable. Use `physical_pin()` (page 222) 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\(^\text{982}\) (the default), the diagram will include ANSI color codes if stdout is a color-capable terminal. Otherwise color can be set to True\(^\text{983}\) or False\(^\text{984}\) 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 251) or `PinMultiplePins` (page 251) 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
- ‘GPIO\(n\)’ where \(n\) is the GPIO number
- ‘WPI\(n\)’ where \(n\) is the wiringPi\(^\text{987}\) pin number
- ‘BCM\(n\)’ where \(n\) is the GPIO number (alias of GPIO\(n\))
- ‘BOARD\(n\)’ where \(n\) is the physical pin number on the main header

---

979 https://docs.python.org/3.7/library/stdtypes.html#str
980 https://docs.python.org/3.7/library/stdtypes.html#set
981 https://docs.python.org/3.7/library/stdtypes.html#str
982 https://docs.python.org/3.7/library/constants.html#None
983 https://docs.python.org/3.7/library/constants.html#True
984 https://docs.python.org/3.7/library/constants.html#False
985 https://docs.python.org/3.7/library/stdtypes.html#str
986 https://docs.python.org/3.7/library/exceptions.html#ValueError
987 http://wiringpi.com/pins/
• ‘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[^988]) 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, of any type.

Note: This does not include the micro-USB or USB-C port used to power the Pi.

usb3
An integer indicating how many of the USB ports are USB3 ports on this revision of the Pi.

ethernet
An integer indicating how many Ethernet ports are physically present on this revision of the Pi.

eth_speed
An integer indicating the maximum speed (in Mbps) of the Ethernet ports (if any). If no Ethernet ports are present, this is 0.

[^988]: https://en.wikipedia.org/wiki/System_on_a_chip
wifi
A bool indicating whether this revision of the Pi has wifi built-in.

bluetooth
A bool indicating whether this revision of the Pi has bluetooth built-in.

csi
An integer indicating the number of CSI (camera) ports available on this revision of the Pi.

dsi
An integer indicating the number of DSI (display) ports available on this revision of the Pi.

headers
A dictionary which maps header labels to `HeaderInfo` (page 224) tuples. For example, to obtain information about header P1 you would query `headers['P1']`. To obtain information about pin 12 on header J8 you would query `headers['J8'].pins[12]`.

A rendered version of this data can be obtained by using the `PiBoardInfo` (page 221) 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 221) object in a format string. For example:

```python
from gpiozero import *
print('{0:board}'.format(pi_info()))
```

### 23.3 HeaderInfo

class gpiozero.HeaderInfo
This class is a `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 codes
print('{0:mono row2}'.format(pi_info().headers['P1'])) # force plain ASCII
```

The following attributes are defined:

**pprint**(color=None)
Pretty-print a diagram of the header pins.

If `color` is `None` (the default, the diagram will include ANSI color codes if stdout is a...
color-capable terminal). Otherwise color can be set to `True` or `False` to force color or monochrome output.

**name**

The name of the header, typically as it appears silk-screened on the board (e.g. “P1” or “J8”).

**rows**

The number of rows on the header.

**columns**

The number of columns on the header.

**pins**

A dictionary mapping physical pin numbers to `PinInfo` tuples.

### 23.4 PinInfo

```python
class gpiozero.PinInfo
```

This class is a `namedtuple` derivative used to represent information about a pin present on a GPIO header. The following attributes are defined:

**number**

An integer containing the physical pin number on the header (starting from 1 in accordance with convention).

**function**

A string describing the function of the pin. Some common examples include “GND” (for pins connecting to ground), “3V3” (for pins which output 3.3 volts), “GPIO9” (for GPIO9 in the Broadcom numbering scheme), etc.

**pull_up**

A bool indicating whether the pin has a physical pull-up resistor permanently attached (this is usually `False` but GPIO2 and GPIO3 are usually `True`). This is used internally by gpiozero to raise errors when pull-down is requested on a pin with a physical pull-up resistor.

**row**

An integer indicating on which row the pin is physically located in the header (1-based)

**col**

An integer indicating in which column the pin is physically located in the header (1-based)

---

992 https://docs.python.org/3.7/library/constants.html#True
993 https://docs.python.org/3.7/library/constants.html#False
994 https://docs.python.org/3.7/library/collections.html#collections.namedtuple
995 https://docs.python.org/3.7/library/constants.html#False
996 https://docs.python.org/3.7/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 232) which turns it into a Pin (page 233) implementation. The default factory can be queried (and changed) with Device.pin_factory (page 203). 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 49) chapter).

This is illustrated in the following flow-chart:
LED(pin_spec,..., pin_factory=None)

pin_factory is None?

yes

Device.pin_factory is None?

yes

self.pin_factory = pin_factory

no

self.pin_factory = Device.pin_factory

no

self.pin = self.pin_factory.pin(pin_spec)

The default factory is constructed when the first device is initialised; 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), a BadPinFactory (page 248) exception will be raised at construction time.

After importing gpiozero, until constructing a gpiozero device, the pin factory is None, but at the point of first construction the default pin factory will come into effect:

```
pi@raspberrypi:- $ python3
Python 3.7.3 (default, Apr 3 2019, 05:39:12)
[ GCC 8.2.0 ) on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> from gpiozero import Device, LED
>>> print(Device.pin_factory)
None

(continues on next page)
```

997 https://docs.python.org/3.7/library/constants.html#None
As above, on a Raspberry Pi with the RPi.GPIO library installed, (assuming no environment variables are set), the default pin factory will be \texttt{RPiGPIOFactory} (page 241).

On a PC (with no pin libraries installed and no environment variables set), importing will work but attempting to create a device will raise \texttt{BadPinFactory} (page 248):

```python
ben@magicman:~ $ python3
Python 3.6.8 (default, Aug 20 2019, 17:12:48)
[GCC 8.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> from gpiozero import Device, LED
>>> print(Device.pin_factory)
None
>>> led = LED(2)
... BadPinFactory: Unable to load any default pin factory!
```

### 24.1 Changing the pin factory

The default pin factory can be replaced by specifying a value for the \texttt{GPIOZERO_PIN_FACTORY} (page 80) environment variable. For example:

```bash
pi@raspberrypi:~ $ GPIOZERO_PIN_FACTORY=native python3
Python 3.7.3 (default, Apr  3 2019, 05:39:12)
[GCC 8.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> from gpiozero import Device
>>> Device._default_pin_factory()
<gpiozero.pins.native.NativeFactory object at 0x762c26b0>
```

To set the \texttt{GPIOZERO_PIN_FACTORY} (page 80) for the rest of your session you can \texttt{export} this value:

```bash
pi@raspberrypi:~ $ export GPIOZERO_PIN_FACTORY=native
pi@raspberrypi:~ $ python3
Python 3.7.3 (default, Apr  3 2019, 05:39:12)
[GCC 8.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import gpiozero
>>> Device._default_pin_factory()
<gpiozero.pins.native.NativeFactory object at 0x762c26b0>
>>> quit()
```

### 24.1. Changing the pin factory
If you add the `export` command to your `~/.bashrc` file, you’ll set the default pin factory for all future sessions too.

If the environment variable is set, the corresponding pin factory will be used, otherwise each of the four GPIO pin factories will be attempted to be used in turn.

The following values, and the corresponding Factory (page 232) and Pin (page 233) 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>rpi-gpio</td>
<td><code>gpiozero.pins.rpigpio.RPiGPIOFactory</code> (page 241)</td>
<td><code>gpiozero.pins.rpigpio.RPiGPIOPin</code> (page 242)</td>
</tr>
<tr>
<td>lgpio</td>
<td><code>gpiozero.pins.lgpio.LGPIOFactory</code> (page 242)</td>
<td><code>gpiozero.pins.lgpio.LGPIOPin</code> (page 242)</td>
</tr>
<tr>
<td>pig-pgio</td>
<td><code>gpiozero.pins.pigpio.PiGPIOFactory</code> (page 242)</td>
<td><code>gpiozero.pins.pigpio.PiGPIOPin</code> (page 243)</td>
</tr>
<tr>
<td>native</td>
<td><code>gpiozero.pins.native.NativeFactory</code> (page 243)</td>
<td><code>gpiozero.pins.native.NativePin</code> (page 243)</td>
</tr>
</tbody>
</table>

If you need to change the default pin factory from within a script, either set `Device.pin_factory` (page 203) to the new factory instance to use:

```python
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:

```python
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.PiGPIOFactory` (page 243) on a remote pi called “remote-pi” you can set the `PIGPIO_ADDR` (page 80) environment variable when running your script:

```
$ GPIOZERO_PIN_FACTORY=pigpio PIGPIO_ADDR=remote-pi python3 my_script.py
```

Like the `GPIOZERO_PIN_FACTORY` (page 80) value, these can be exported from your `~/.bashrc` script too.

**Warning:** The astute and mischievous reader may note that it is possible to mix factories, e.g. using `RPiGPIOFactory` (page 241) for one pin, and `NativeFactory` (page 243) for another. This is unsupported, and if it results in your script crashing, your components failing, or your Raspberry Pi turning into an actual raspberry pie, you have only yourself to blame.

Sensible uses of multiple pin factories are given in *Configuring Remote GPIO* (page 49).
24.2 Mock pins

There’s also a MockFactory (page 244) 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 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, either by setting an environment variable or setting the pin factory within the script. For example:

```
pi@raspberrypi:~ $ GPIOZERO_PIN_FACTORY=mock python3
```

or:

```
from gpiozero import Device, LED
from gpiozero.pins.mock import MockFactory

Device.pin_factory = MockFactory()
led = LED(2)
```

You can create device objects and inspect their value changing as you’d expect:

```
pi@raspberrypi:~ $ GPIOZERO_PIN_FACTORY=mock python3
Python 3.7.3 (default, Apr 3 2019, 05:39:12)
[GCC 8.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> from gpiozero import LED
>>> led = LED(2)
>>> led.value
0
>>> led.on()
>>> led.value
1
```

You can even control pin state changes to simulate device behaviour:

```
>>> from gpiozero import LED, Button

# 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

# Initially the button isn't "pressed" so the LED should be off
>>> led.value
0

# Drive the pin low (this is what would happen electrically when the button is pressed)
>>> btn.pin.drive_low()
# The LED is now on
>>> led.value
1

>>> btn.pin.drive_high()
# The button is now "released", so the LED should be off again
>>> led.value
0
```
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), for example, you have to use `MockPWMPin` (page 244) to be able to use devices requiring PWM:

```
pi@raspberrypi:~ $ GPIOZERO_PIN_FACTORY=mock GPIOZERO_MOCK_PIN_CLASS=mockpwmpin
...python3
```

or:

```
from gpiozero import Device, LED
from gpiozero.pins.mock import MockFactory, MockPWMPin

Device.pin_factory = MockFactory(pin_class=MockPWMPin)

led = LED(2)
```

Interested users are invited to read the GPIO Zero test suite\(^998\) for further examples of usage.

### 24.3 Base classes

#### class `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 233)
- `ticks_diff()` (page 233)

Descendents may override the following methods, if applicable:

- `close()` (page 232)
- `reserve_pins()` (page 233)
- `release_pins()` (page 232)
- `release_all()` (page 232)
- `pin()` (page 232)
- `spi()` (page 233)
- `_get_pi_info()`

**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 233) descendent representing the specified pin.

**Warning:** Descendents must ensure that pin instances representing the same hardware are identical; i.e. two separate invocations of `pin()` (page 232) 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 232) for further information.

\(^998\) https://github.com/gpiozero/gpiozero/tree/master/tests
**release_pins(reserver, *pins)***
Releases the reservation of *reserver* against *pins*. This is typically called during `close()` (page 203) 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 232).

**spi(****spi_args)**
Returns an instance of an SPI (page 236) 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 249).

**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 233) method.

**ticks_diff(later, earlier)**
Return the time in seconds between two `ticks()` (page 233) 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 233).

**pi_info**
Returns a `PiBoardInfo` (page 221) 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`.⁹⁹⁹

---

⁹⁹⁹ https://docs.python.org/3.7/library/constants.html#None

---

24.3. Base classes 233
close()
Cleans up the resources allocated to the pin. After this method is called, this *Pin* (page 233) 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 235) is currently “rising”, **bounce** (page 234) is currently 5/1000 (5ms), then the waveform below will only fire **when_changed** (page 236) on two occasions despite there being three rising edges:

```
TIME 0...1...2...3...4...5...6...7...8...9...10...11...12 ms
bounce elimination |===================| |==============
HIGH - - - - > ,--. ,--------------. ,--.
| | | | | | | |
| | | | | | | |
LOW ----------------' `--' `-' `-----------
: : :
when_changed when_changed
fires fires
```

---

1000 https://docs.python.org/3.7/library/constants.html#None
If the pin does not support edge detection, attempts to set this property will raise `PinEdgeDetectUnsupported` (page 250). 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 236). This can be one of the strings “both” (the default), “rising”, “falling”, or “none”:

```
HIGH - - - - > ,----------------------
| | |
| | |
| : : :
| : : :
LOW -----------------' `--------------
```

Fires when `changed` "both" "both"
when edges is ... "rising" "falling"

If the pin does not support edge detection, attempts to set this property will raise `PinEdgeDetectUnsupported` (page 250).

**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 251) 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 250) 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 250). If the specified value is not supported by the underlying hardware, `PinInvalidPull` (page 250) 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 235) for more information):

```
HIGH - - - - > ,----------------------
| |
| |
LOW -----------------' `--------------
```

Descendants which implement analog, or analog-like capabilities can return values between 0 and 1. For example, pins implementing PWM (where `frequency` (page 235) is not `None`) return a value between 0.0 and 1.0 representing the current PWM duty cycle.

---

1001 https://docs.python.org/3.7/library/constants.html#None
1002 https://docs.python.org/3.7/library/constants.html#None
1003 https://docs.python.org/3.7/library/constants.html#None
1004 https://docs.python.org/3.7/library/constants.html#None
If a pin is currently configured for input, and an attempt is made to set this attribute, \texttt{PinSetInput} (page 250) will be raised. If an invalid value is specified for this attribute, \texttt{PinInvalidState} (page 250) will be raised.

\textbf{when\_changed}

A function or bound method to be called when the pin’s state changes (more specifically when the edge specified by \texttt{edges} (page 235) is detected on the pin). The function or bound method must accept two parameters: the first will report the ticks (from \texttt{Factory.ticks()} (page 233)) when the pin’s state changed, and the second will report the pin’s current state.

\begin{Verbatim}
\textbf{Warning:} Depending on hardware support, the state is \textbf{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 \textit{since} the interrupt. Exercise appropriate caution when relying upon this parameter.
\end{Verbatim}

If the pin does not support edge detection, attempts to set this property will raise \texttt{PinEdgeDetectUnsupported} (page 250).

\begin{Verbatim}
\textbf{class} \texttt{gpiozero.SPI(\texttt{\textasteriskcentered}, \texttt{pin\_factory=None})}
\end{Verbatim}

Abstract interface for \texttt{Serial Peripheral Interface} \texttt{1005} (SPI) implementations. Descendents \textbf{must} override the following methods:

\begin{itemize}
  \item \texttt{transfer()} (page 236)
  \item \texttt{\_get\_clock\_mode()}
\end{itemize}

Descendents \textbf{may} override the following methods, if applicable:

\begin{itemize}
  \item \texttt{read()} (page 236)
  \item \texttt{write()} (page 236)
  \item \texttt{\_set\_clock\_mode()}
  \item \texttt{\_get\_lsb\_first()}
  \item \texttt{\_set\_lsb\_first()}
  \item \texttt{\_get\_select\_high()}
  \item \texttt{\_set\_select\_high()}
  \item \texttt{\_get\_bits\_per\_word()}
  \item \texttt{\_set\_bits\_per\_word()}
\end{itemize}

\begin{Verbatim}
\texttt{read(n)}
\end{Verbatim}

Read \texttt{n} words of data from the SPI interface, returning them as a sequence of unsigned ints, each no larger than the configured \texttt{bits\_per\_word} (page 237) of the interface.

This method is typically used with read-only devices that feature half-duplex communication. See \texttt{transfer()} (page 236) for full duplex communication.

\begin{Verbatim}
\texttt{transfer(data)}
\end{Verbatim}

Write \texttt{data} to the SPI interface. \texttt{data} must be a sequence of unsigned integer words each of which will fit within the configured \texttt{bits\_per\_word} (page 237) 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 \texttt{data} written to the interface. Each word in the returned sequence will be an unsigned integer no larger than the configured \texttt{bits\_per\_word} (page 237) of the interface.

\begin{footnotesize}
\footnotesize
\begin{itemize}
  \item \texttt{1005} https://en.wikipedia.org/wiki/Serial_Peripheral_Interface_Bus
\end{itemize}
\end{footnotesize}
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 237) 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 236) 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 238) and clock_phase (page 237) 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 238) and clock_phase (page 237) 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 attribute (corresponding to CPOL).

The following diagram indicates when data is read when clock_polarity (page 238) is False, and clock_phase (page 237) is False (the default), equivalent to CPHA 0:

```
CLK | | | | | | | | | | | | | |
    | | | | | | | | | | | | | |
-----' `---' `---' `---' `---' `---' `---' `-------
MISO---. ,---. ,---. ,---. ,---. ,---. ,---. ,---.
/ \
/ \
/ \
/ \
/ \
/ \
/ \
/ \
/ \
/ \
/ \
/ \
/ \
/ \\---' `---' `---' `---' `---' `---' `---'

The following diagram indicates when data is read when clock_polarity (page 238) is False, but clock_phase (page 237) is True, equivalent to CPHA 1:

---. ,---. ,---. ,---. ,---. ,---. ,---. ,---.
/ \
/ \
/ \
/ \
/ \
/ \
/ \
/ \
/ \
/ \
/ \
/ \
/ \
/ \\---' `---' `---' `---' `---' `---' `---'

https://docs.python.org/3.7/library/constants.html#False
https://docs.python.org/3.7/library/constants.html#True
https://docs.python.org/3.7/library/constants.html#False
https://docs.python.org/3.7/library/constants.html#False
https://docs.python.org/3.7/library/constants.html#False
https://docs.python.org/3.7/library/constants.html#True
### clock_polarity

The polarity of the SPI clock pin. If this is `False` (the default), the clock pin will idle low, and pulse high. Setting this to `True` will cause the clock pin to idle high, and pulse low. On many data sheets this is documented as the CPOL value.

The following diagram illustrates the waveform when `clock_polarity` (page 238) is `False` (the default), equivalent to CPOL 0:

```
<table>
<thead>
<tr>
<th>on</th>
<th>on</th>
<th>on</th>
<th>on</th>
<th>on</th>
<th>on</th>
<th>on</th>
<th>on</th>
</tr>
</thead>
</table>
CLK |    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|
idle | off | off | off | off | off | off | off | idle
```

The following diagram illustrates the waveform when `clock_polarity` (page 238) is `True`, equivalent to CPOL 1:

```
<table>
<thead>
<tr>
<th>idle</th>
<th>off</th>
<th>off</th>
<th>off</th>
<th>off</th>
<th>off</th>
<th>off</th>
<th>idle</th>
</tr>
</thead>
</table>
CLK |     |     |     |     |     |     |     |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>on</td>
<td>on</td>
<td>on</td>
<td>on</td>
<td>on</td>
<td>on</td>
<td>on</td>
<td>on</td>
</tr>
</tbody>
</table>
```

### 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 the a word containing the number 5 (binary 0101) transmitted on MISO with `bits_per_word` (page 237) set to 4, and `clock_mode` (page 237) set to 0, when `lsb_first` (page 238) is `False` (the default):

```
CLK |   |   |   |   |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
</table>
<pre><code>|   |   |   |
</code></pre>
MISO: | : | : | : |
| : | : | : |
```

---

1012 https://docs.python.org/3.7/library/constants.html#False
1013 https://docs.python.org/3.7/library/constants.html#True
1014 https://docs.python.org/3.7/library/constants.html#False
1015 https://docs.python.org/3.7/library/constants.html#True
1016 https://docs.python.org/3.7/library/constants.html#False
1017 https://en.wikipedia.org/wiki/Endianness#Bit_endianness
1018 https://docs.python.org/3.7/library/constants.html#False
And now with `lsb_first` (page 238) set to `True` (and all other parameters the same):

```
  CL | | | | | | | |
  | | | | | | | |
  ----' `---' `---' `---' `---'
MISO: | | | | | | | |
  | | | | | | | |
  --' : `-------' : `-------'
LSB  MSB
```

**rate**

Controls the speed of the SPI interface in Hz (or baud).

Note that most software SPI implementations ignore this property, and will raise `SPIFixedRate` if an attempt is made to set it, as they have no rate control (they simply bit-bang as fast as possible because typically this isn’t very fast anyway, and introducing measures to limit the rate would simply slow them down to the point of being useless).

**select_high**

If `False` (the default), the chip select line is considered active when it is pulled low. When set to `True`, the chip select line is considered active when it is driven high.

The following diagram shows the waveform of the chip select line, and the clock when `clock_polarity` (page 238) is `False`, and `select_high` (page 239) is `False` (the default):

```
  --- |    ,------
CS | chip is selected, and will react to clock | idle
    '-----------------------------------------------------'
CLK | | | | | | | |
  | | | | | | | |
  ----' `---' `---' `---' `---' `---' `---' `---'
```

And when `select_high` (page 239) is `True`:

```
  '-----------------------------------------------------'
CS | chip is selected, and will react to clock | idle
  | | | | | | | |
  | | | | | | | |
  ---' `------'
CLK | | | | | | | |
  | | | | | | | |
```

1019 https://docs.python.org/3.7/library/constants.html#True
1020 https://docs.python.org/3.7/library/constants.html#False
1021 https://docs.python.org/3.7/library/constants.html#True
1022 https://docs.python.org/3.7/library/constants.html#False
1023 https://docs.python.org/3.7/library/constants.html#False
1024 https://docs.python.org/3.7/library/constants.html#True
class gpiozero.pins.pi.PiFactory
Extends Factory (page 232). Abstract base class representing hardware attached to a Raspberry Pi. This forms the base of LocalPiFactory (page 241).

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 descendent representing the specified pin.

**Warning:** Descendents must ensure that pin instances representing the same hardware are identical; i.e. two separate invocations of pin() (page 240) for the same pin specification must return the same object.

release_pins(reserver, *pins)
Releases the reservation of reserver against pins. This is typically called during close() (page 203) 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 240).

spi(**spi_args)
Returns an SPI interface, for the specified SPI port and device, or for the specified pins (clock_pin, mosi_pin, miso_pin, and select_pin). Only one of the schemes can be used; attempting to mix port and device with pin numbers will raise SPIBadArgs (page 249).

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 hardwarebasedinterface(usingspidev)willbereturned. 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 simpler devices this doesn’t matter).

class gpiozero.pins.pi.PiPin(factory, number)
Extends Pin (page 233). 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()

class gpiozero.pins.local.LocalPiFactory
Extends PiFactory (page 240). Abstract base class representing pins attached locally to a Pi. This forms the base class for local-only pin interfaces (RPiGPIOPin (page 242), LGPIOPin (page 242), and NativePin (page 243)).

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 241) method.

static ticks_diff(later, earlier)
Return the time in seconds between two ticks() (page 241) 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 241).

class gpiozero.pins.local.LocalPiPin(factory, number)
Extends PiPin (page 240). Abstract base class representing a multi-function GPIO pin attached to the local Raspberry Pi.

24.4 RPi.GPIO

class gpiozero.pins.rpigpio.RPiGPIOFactory
Extends LocalPiFactory (page 241). Uses the RPi.GPIO library to interface to the Pi’s GPIO pins. This is the default pin implementation if the RPi.GPIO library is installed. Supports all features including PWM (via software).

Because this is the default pin implementation you can use it simply by specifying an integer number for the pin in most operations, e.g.:

```python
from gpiozero import LED

led = LED(12)
```

However, you can also construct RPi.GPIO pins manually if you wish:

---

1025 https://pypi.python.org/pypi/RPi.GPIO
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 241). Pin implementation for the RPi.GPIO\textsuperscript{1026} library. See RPiGPIOFactory (page 241) for more information.

24.5 lgpio

class gpiozero.pins.lgpio.LGPIOFactory(chip=0)
    Extends LocalPiFactory (page 241). Uses the lgpio\textsuperscript{1027} library to interface to the local computer’s GPIO pins. The lgpio library simply talks to Linux gpiochip devices; it is not specific to the Raspberry Pi although this class is currently constructed under the assumption that it is running on a Raspberry Pi.

    You can construct lgpio pins manually like so:

        from gpiozero.pins.lgpio import LGPIOFactory
        from gpiozero import LED
        factory = LGPIOFactory(chip=0)
        led = LED(12, pin_factory=factory)

    The \texttt{chip} parameter to the factory constructor specifies which gpiochip device to attempt to open. It defaults to 0 and thus doesn’t normally need to be specified (the example above only includes it for completeness).

    The lgpio library relies on access to the \texttt{/dev/gpiochip*} devices. If you run into issues, please check that your user has read/write access to the specific gpiochip device you are attempting to open (0 by default).

class gpiozero.pins.lgpio.LGPIOPin(factory, number)
    Extends LocalPiPin (page 241). Pin implementation for the lgpio\textsuperscript{1028} library. See LGPIOFactory (page 242) for more information.

24.6 PiGPIO

class gpiozero.pins.pigpio.PiGPIOFactory(host=None, port=None)
    Extends PiFactory (page 240). Uses the pigpio\textsuperscript{1029} library to interface to the Pi’s GPIO pins. The pigpio library relies on a daemon (pigpiod) to be running as root to provide access to the GPIO pins, and communicates with this daemon over a network socket.

    While this does mean only the daemon itself should control the pins, the architecture does have several advantages:

    \begin{itemize}
    \item 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\textsuperscript{1030} client library installed on it)
    \item The daemon supports hardware PWM via the DMA controller
    \end{itemize}
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.

```python
class gpiozero.pins.pigpio.PiGPIOPin(factory, number)
Extends PiPin (page 240). Pin implementation for the pigpio library. See PiGPIOFactory (page 242) for more information.
```

### 24.7 Native

```python
class gpiozero.pins.native.NativeFactory
Extends LocalPiFactory (page 241). 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)
```

```python
class gpiozero.pins.native.NativePin(factory, number)
Extends NativePin (page 243) for Pi hardware prior to the Pi 4 (Pi 0, 1, 2, 3, and 3+).
```

1031 http://abyz.me.uk/rpi/pigpio/
class gpiozero.pins.native.Native2711Pin(factory, number)
    Extends NativePin (page 243) for Pi 4 hardware (Pi 4, CM4, Pi 400 at the time of writing).

24.8 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 Factory.pi_info attribute as well as where pull-ups are assumed to be).
    
    The pin_class attribute specifies which mock pin class will be generated by the pin() (page 244) method by default. This can be changed after construction by modifying the pin_class (page 244) attribute.

    pin(spec, pin_class=None, **kwargs)
        The pin method for MockFactory (page 244) additionally takes a pin_class attribute which can be used to override the class' pin_class (page 244) attribute. Any additional keyword arguments will be passed along to the pin constructor (useful with things like MockConnectedPin (page 244) 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.

    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 244) method.

    static ticks_diff(later, earlier)
        Return the time in seconds between two ticks() (page 244) 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 244).

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 244) adds PWM support.

class gpiozero.pins.mock.MockConnectedPin(factory, number, input_pin=None)
    This derivative of MockPin (page 244) 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 244) 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 244) is intended to be used with another MockPin (page 244)
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.

```python
class gpiozero.pins.mock.MockSPIDevice(clock_pin, mosi_pin=None, miso_pin=None,
                                       select_pin=None, *, clock_polarity=False,
                                       clock_phase=False, lsb_first=False,
                                       bits_per_word=8, select_high=False,
                                       pin_factory=None)
```

This class is used to test `SPIDevice` implementations. It can be used to mock up the slave side of simple SPI devices, e.g. the MCP3xxx series of ADCs.

Descendants should override the `on_start()` and/or `on_bit()` methods to respond to SPI interface events. The `rx_word()` and `tx_word()` methods can be used facilitate communications within these methods. Such descendents can then be passed as the `spi_class` parameter of the `MockFactory` (page 244) constructor to have instances attached to any SPI interface requested by an `SPIDevice`. 
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 247), 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 147) you would expect a `ValueError` to be raised. In fact, a `OutputDeviceBadValue` (page 250) error will be raised. However, note that this descends from both `GPIOZeroError` (page 247) (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')
```

### 25.1 Errors

**exception** `gpiozero.GPIOZeroError`

Bases: `Exception`

Base class for all exceptions in GPIO Zero

---

1032 [https://docs.python.org/3.7/library/exceptions.html#ValueError](https://docs.python.org/3.7/library/exceptions.html#ValueError)
1033 [https://docs.python.org/3.7/library/exceptions.html#ValueError](https://docs.python.org/3.7/library/exceptions.html#ValueError)
1034 [https://docs.python.org/3.7/library/exceptions.html#Exception](https://docs.python.org/3.7/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 203)

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 182)

1035 https://docs.python.org/3.7/library/exceptions.html#ValueError
1036 https://docs.python.org/3.7/library/exceptions.html#ValueError
1037 https://docs.python.org/3.7/library/exceptions.html#ValueError
1038 https://docs.python.org/3.7/library/exceptions.html#ImportError
1039 https://docs.python.org/3.7/library/exceptions.html#RuntimeError
1040 https://docs.python.org/3.7/library/exceptions.html#ValueError
1041 https://docs.python.org/3.7/library/exceptions.html#ValueError
1042 https://docs.python.org/3.7/library/exceptions.html#ValueError
1043 https://docs.python.org/3.7/library/exceptions.html#ValueError
1044 https://docs.python.org/3.7/library/exceptions.html#ValueError
exception gpiozero.SPIError
Bases: gpiozero.exc.GPIOZeroError
Base class for errors related to the SPI implementation

exception gpiozero.SPIBadArgs
Bases: gpiozero.exc.SPIError, ValueError\(^{1045}\)
Error raised when invalid arguments are given while constructing SPIDevice (page 156)

exception gpiozero.SPIBadChannel
Bases: gpiozero.exc.SPIError, ValueError\(^{1046}\)
Error raised when an invalid channel is given to an AnalogInputDevice (page 155)

exception gpiozero.SPIFixedClockMode
Bases: gpiozero.exc.SPIError, AttributeError\(^{1047}\)
Error raised when the SPI clock mode cannot be changed

exception gpiozero.SPIInvalidClockMode
Bases: gpiozero.exc.SPIError, ValueError\(^{1048}\)
Error raised when an invalid clock mode is given to an SPI implementation

exception gpiozero.SPIFixedBitOrder
Bases: gpiozero.exc.SPIError, AttributeError\(^{1049}\)
Error raised when the SPI bit-endianness cannot be changed

exception gpiozero.SPIFixedSelect
Bases: gpiozero.exc.SPIError, AttributeError\(^{1050}\)
Error raised when the SPI select polarity cannot be changed

exception gpiozero.SPIFixedWordSize
Bases: gpiozero.exc.SPIError, AttributeError\(^{1051}\)
Error raised when the number of bits per word cannot be changed

exception gpiozero.SPIInvalidWordSize
Bases: gpiozero.exc.SPIError, ValueError\(^{1052}\)
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 247)

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\(^{1053}\)

\(^{1045}\) https://docs.python.org/3.7/library/exceptions.html#ValueError
\(^{1046}\) https://docs.python.org/3.7/library/exceptions.html#ValueError
\(^{1047}\) https://docs.python.org/3.7/library/exceptions.html#AttributeError
\(^{1048}\) https://docs.python.org/3.7/library/exceptions.html#ValueError
\(^{1049}\) https://docs.python.org/3.7/library/exceptions.html#AttributeError
\(^{1050}\) https://docs.python.org/3.7/library/exceptions.html#AttributeError
\(^{1051}\) https://docs.python.org/3.7/library/exceptions.html#AttributeError
\(^{1052}\) https://docs.python.org/3.7/library/exceptions.html#ValueError
\(^{1053}\) https://docs.python.org/3.7/library/exceptions.html#ValueError
Error raised when a pin specification is not given

```
exception gpiozero.InputDeviceError
    Bases: gpiozero.exc.GPIODeviceError
    Base class for errors specific to the InputDevice hierarchy

exception gpiozero.OutputDeviceError
    Bases: gpiozero.exc.GPIODeviceError
    Base class for errors specified to the OutputDevice hierarchy

exception gpiozero.OutputDeviceBadValue
    Bases: gpiozero.exc.OutputDeviceError, ValueError
    Error raised when value is set to an invalid value

exception gpiozero.PinError
    Bases: gpiozero.exc.GPIOZeroError
    Base class for errors related to pin implementations

exception gpiozero.PinInvalidFunction
    Bases: gpiozero.exc.PinError, ValueError
    Error raised when attempting to change the function of a pin to an invalid value

exception gpiozero.PinInvalidState
    Bases: gpiozero.exc.PinError, ValueError
    Error raised when attempting to assign an invalid state to a pin

exception gpiozero.PinInvalidPull
    Bases: gpiozero.exc.PinError, ValueError
    Error raised when attempting to assign an invalid pull-up to a pin

exception gpiozero.PinInvalidEdges
    Bases: gpiozero.exc.PinError, ValueError
    Error raised when attempting to assign an invalid edge detection to a pin

exception gpiozero.PinInvalidBounce
    Bases: gpiozero.exc.PinError, ValueError
    Error raised when attempting to assign an invalid bounce time to a pin

exception gpiozero.PinSetInput
    Bases: gpiozero.exc.PinError, AttributeError
    Error raised when attempting to set a read-only pin

exception gpiozero.PinFixedPull
    Bases: gpiozero.exc.PinError, AttributeError
    Error raised when attempting to set the pull of a pin with fixed pull-up

exception gpiozero.PinEdgeDetectUnsupported
    Bases: gpiozero.exc.PinError, AttributeError
    Error raised when attempting to use edge detection on unsupported pins
```
exception gpiozero.PinUnsupported
    Bases: gpiozero.exc.PinError, NotImplementedErrort
    Error raised when attempting to obtain a pin interface on unsupported pins

exception gpiozero.PinSPIUnsupported
    Bases: gpiozero.exc.PinError, NotImplementedErrort
    Error raised when attempting to obtain an SPI interface on unsupported pins

exception gpiozero.PinPWMError
    Bases: gpiozero.exc.PinError
    Base class for errors related to PWM implementations

exception gpiozero.PinPWMUnsupported
    Bases: gpiozero.exc.PinPWMErrort, AttributeError
    Error raised when attempting to activate PWM on unsupported pins

exception gpiozero.PinPWMMutableValue
    Bases: gpiozero.exc.PinPWMErrort, AttributeError
    Error raised when attempting to initialize PWM on an input pin

exception gpiozero.PinUnknownPi
    Bases: gpiozero.exc.PinErrort, RuntimeError
    Error raised when gpiozero doesn’t recognize a revision of the Pi

exception gpiozero.PinMultiplePins
    Bases: gpiozero.exc.PinErrort, RuntimeError
    Error raised when multiple pins support the requested function

exception gpiozero.PinNoPins
    Bases: gpiozero.exc.PinErrort, RuntimeError
    Error raised when no pins support the requested function

exception gpiozero.PinInvalidPin
    Bases: gpiozero.exc.PinErrort, ValueError
    Error raised when an invalid pin specification is provided

25.2 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

References:

1063 https://docs.python.org/3.7/library/exceptions.html#NotImplementedError
1064 https://docs.python.org/3.7/library/exceptions.html#NotImplementedError
1065 https://docs.python.org/3.7/library/exceptions.html#AttributeError
1066 https://docs.python.org/3.7/library/exceptions.html#AttributeError
1067 https://docs.python.org/3.7/library/exceptions.html#RuntimeError
1068 https://docs.python.org/3.7/library/exceptions.html#RuntimeError
1069 https://docs.python.org/3.7/library/exceptions.html#RuntimeError
1070 https://docs.python.org/3.7/library/exceptions.html#ValueError
1071 https://docs.python.org/3.7/library/exceptions.html#Warning
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 SPI 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
26.1 Release 2.0 (???)

- Removed Python 2.x support; many thanks to Fangchen Li for a substantial amount of work on this! (#799, #896)  
- Removed RPIO pin implementation  
- Added Backwards Compatibility (page 89) chapter

26.2 Release 1.6.2 (2021-03-18)

- Correct docs referring to 1.6.0 as the last version supporting Python 2

**Warning:** This is the last release to support Python 2

26.3 Release 1.6.1 (2021-03-17)

- Fix missing font files for 7-segment displays

26.4 Release 1.6.0 (2021-03-14)

- Added RotaryEncoder (page 118) class (thanks to Paulo Mateus) (#482, #928)

---

1072 https://github.com/gpiozero/gpiozero/issues/799  
1073 https://github.com/gpiozero/gpiozero/issues/896  
1074 https://github.com/gpiozero/gpiozero/issues/482  
1075 https://github.com/gpiozero/gpiozero/issues/928
• Added support for multi-segment character displays with `LEDCharDisplay` (page 164) and `LEDMultiCharDisplay` (page 166) along with “font” support using `LEDCharFont` (page 167) (thanks to Martin O’Hanlon) (#357\(^{1076}\), #485\(^{1077}\), #488\(^{1078}\), #493\(^{1079}\), #930\(^{1080}\))

• Added `Pibrella` (page 178) class (thanks to Carl Monk) (#773\(^{1081}\), #798\(^{1082}\))

• Added `TrafficpHat` (page 176) class (thanks to Ryan Walmsley) (#845\(^{1083}\), #846\(^{1084}\))

• Added support for the `lgpio` library as a pin factory (`LGPIOFactory` (page 242)) (thanks to Joan for lg) (#927\(^{1085}\))

• Allow `Motor` (page 137) to pass `pin_factory` (page 203) to its child `OutputDevice` (page 146) objects (thanks to Yisrael Dov Lebow) (#792\(^{1087}\))

• Small SPI exception fix (thanks to Maksim Levental) (#762\(^{1088}\))

• Warn users when using default pin factory for Servos and Distance Sensors (thanks to Sofiia Kosovan and Daniele Procida at the EuroPython sprints) (#780\(^{1089}\), #781\(^{1090}\))

• Added `pulse_width` (page 141) property to `Servo` (page 139) (suggested by Daniele Procida at the PyCon UK sprints) (#795\(^{1091}\), #797\(^{1092}\))

• Added event-driven functionality to `internal devices` (page 191) (#941\(^{1093}\))

• Allowed `Energenie` (page 182) sockets preserve their state on construction (thanks to Jack Warden) (#865\(^{1094}\))

• Added source tools `scaled_half()` and `scaled_full()`

• Added complete Pi 4 support to `NativeFactory` (page 243) (thanks to Andrew Scheller) (#920\(^{1095}\), #926\(^{1096}\), #940\(^{1097}\))

• Updated add-on boards to use BOARD numbering (#349\(^{1098}\), #860\(^{1099}\))

• Fixed `ButtonBoard` (page 167) release events (#761\(^{1100}\))

• Add ASCII art diagrams to `pinout` for Pi 400 and CM4 (#39\(^{1101}\))

• Cleaned up software SPI (thanks to Andrew Scheller and Kyle Morgan) (#777\(^{1102}\), #895\(^{1103}\), #900\(^{1104}\))

• Added USB3 and Ethernet speed attributes to `pi_info()` (page 221)

---

\(^{1076}\) https://github.com/gpiozero/gpiozero/issues/357
\(^{1077}\) https://github.com/gpiozero/gpiozero/issues/485
\(^{1078}\) https://github.com/gpiozero/gpiozero/issues/488
\(^{1079}\) https://github.com/gpiozero/gpiozero/issues/493
\(^{1080}\) https://github.com/gpiozero/gpiozero/issues/930
\(^{1081}\) https://github.com/gpiozero/gpiozero/issues/773
\(^{1082}\) https://github.com/gpiozero/gpiozero/issues/798
\(^{1083}\) https://github.com/gpiozero/gpiozero/issues/845
\(^{1084}\) https://github.com/gpiozero/gpiozero/issues/846
\(^{1085}\) http://abyz.me.uk/lc/py_lgpio.html
\(^{1086}\) https://github.com/gpiozero/gpiozero/issues/927
\(^{1087}\) https://github.com/gpiozero/gpiozero/issues/929
\(^{1088}\) https://github.com/gpiozero/gpiozero/issues/762
\(^{1089}\) https://github.com/gpiozero/gpiozero/issues/780
\(^{1090}\) https://github.com/gpiozero/gpiozero/issues/795
\(^{1091}\) https://github.com/gpiozero/gpiozero/issues/797
\(^{1092}\) https://github.com/gpiozero/gpiozero/issues/941
\(^{1093}\) https://github.com/gpiozero/gpiozero/issues/865
\(^{1094}\) https://github.com/gpiozero/gpiozero/issues/920
\(^{1095}\) https://github.com/gpiozero/gpiozero/issues/929
\(^{1096}\) https://github.com/gpiozero/gpiozero/issues/940
\(^{1097}\) https://github.com/gpiozero/gpiozero/issues/349
\(^{1098}\) https://github.com/gpiozero/gpiozero/issues/860
\(^{1099}\) https://github.com/gpiozero/gpiozero/issues/761
\(^{1100}\) https://github.com/gpiozero/gpiozero/issues/932
\(^{1101}\) https://github.com/gpiozero/gpiozero/issues/777
\(^{1102}\) https://github.com/gpiozero/gpiozero/issues/895
\(^{1103}\) https://github.com/gpiozero/gpiozero/issues/900
26.5 Release 1.5.1 (2019-06-24)

- Added Raspberry Pi 4 data for `pi_info()` (page 221) and `pinout`
- Minor docs updates

26.6 Release 1.5.0 (2019-02-12)

- Introduced pin event timing to increase accuracy of certain devices such as the HC-SR04 `DistanceSensor` (page 115). (#664, #665)
- Further improvements to `DistanceSensor` (page 115) (ignoring missed edges). (#719)
- Allow `source` to take a device object as well as `values` or other `values`. See `Source/Values` (page 65). (#640)
- Added internal device classes `LoadAverage` (page 196) and `DiskUsage` (page 197) (thanks to Jeven M R for the latter). (#532, #714)
- Added support for `colorzero` with `RGBLED` (page 131) (this adds a new dependency). (#655)
- Added `TonalBuzzer` (page 135) with `Tone` (page 219) API for specifying frequencies raw or via MIDI or musical notes. (#681, #717)
- Added `PiHutXmasTree` (page 171). (#502)
- Added `PumpkinPi` (page 186) and `JamHat` (page 177) (thanks to Claire Pollard). (#680, #681, #717)
- Ensured gpiozero can be imported without a valid pin factory set. (#591, #713)
- Reduced import time by not computing default pin factory at the point of import. (#675, #722)
- Added support for various pin numbering mechanisms. (#470)
- `Motor` (page 137) instances now use `DigitalOutputDevice` (page 143) for non-PWM pins.
- Allow non-PWM use of `Robot` (page 179). (#481)
- Added optional `enable` init param to `Motor` (page 137). (#366)
• Added --xyz option to pinout command line tool to open pinout.xyz in a web browser. (#604)

• Added 3B+, 3A+ and CM3+ to Pi model data. (#627, #704)

• Minor improvements to Energenie (page 182), thanks to Steve Amor. (#629, #634)

• Allow SmoothedInputDevice (page 123), LightSensor (page 113) and MotionSensor (page 112) to have pull-up configured. (#652)

• Allow input devices to be pulled up or down externally, thanks to Philippe Muller. (#593, #658)

• Minor changes to support Python 3.7, thanks to Russel Winder and Rick Ansell. (#666, #668, #669, #671, #673)

• Added zip_values() (page 212) source tool.

• Correct row/col numbering logic in PinInfo (page 225). (#674)

• 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.

26.7 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 180) and Robot.backward() (page 179). (#306 and #619)

• Fixed DistanceSensor (page 115) 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 138), PhaseEnableRobot, and descendants, thanks to Ian Harcombe! (#386)

https://pinout.xyz
https://github.com/gpiozero/gpiozero/issues/604
https://github.com/gpiozero/gpiozero/issues/627
https://github.com/gpiozero/gpiozero/issues/704
https://github.com/gpiozero/gpiozero/issues/629
https://github.com/gpiozero/gpiozero/issues/634
https://github.com/gpiozero/gpiozero/issues/652
https://github.com/gpiozero/gpiozero/issues/593
https://github.com/gpiozero/gpiozero/issues/658
https://github.com/gpiozero/gpiozero/issues/666
https://github.com/gpiozero/gpiozero/issues/668
https://github.com/gpiozero/gpiozero/issues/669
https://github.com/gpiozero/gpiozero/issues/671
https://github.com/gpiozero/gpiozero/issues/673
https://github.com/gpiozero/gpiozero/issues/674
https://github.com/gpiozero/gpiozero/issues/306
https://github.com/gpiozero/gpiozero/issues/619
https://github.com/gpiozero/gpiozero/issues/584
https://github.com/gpiozero/gpiozero/issues/595
https://github.com/gpiozero/gpiozero/issues/617
https://github.com/gpiozero/gpiozero/issues/618
https://github.com/gpiozero/gpiozero/issues/386
A variety of other minor enhancements, largely thanks to Andrew Scheller! (#479, #489, #491, #492)

26.8 Release 1.4.0 (2017-07-26)

- Pin factory is now configurable from device constructors (page 229) as well as command line. NOTE: this is a backwards incompatible change for manual pin construction but it’s hoped this is (currently) a sufficiently rare use case that this won’t affect too many people and the benefits of the new system warrant such a change, i.e. the ability to use remote pin factories with HAT classes that don’t accept pin assignments (#279)
- 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 65), better charts, more recipes, remote GPIO configuration (page 49), mock pins, better PDF output (#484, #469, #523, #529, #565, #576)
- Support for StatusZero (page 183) and StatusBoard (page 184) HATs (#558)
- Added pinout command line tool to provide a simple reference to the GPIO layout and information about the associated Pi thanks to Stewart Adcock for the initial work pi_info() (page 221) made more lenient for new (unknown) Pi models (#529)
- Fixed a variety of packaging issues (#535, #518, #519)
- Improved text in factory fallback warnings (#572)

26.9 Release 1.3.2 (2017-03-03)

- Added new Pi models to stop pi_info() (page 221) breaking
- Fix issue with pi_info() (page 221) breaking on unknown Pi models
26.10 Release 1.3.1 (2016-08-31 ... later)

- Fixed hardware SPI support which Dave broke in 1.3.0. Sorry!
- Some minor docs changes

26.11 Release 1.3.0 (2016-08-31)

- Added `ButtonBoard` (page 167) for reading multiple buttons in a single class (#340)
- Added `Servo` (page 139) and `AngularServo` (page 141) classes for controlling simple servo motors (#248)
- Lots of work on supporting easier use of internal and third-party pin implementations (#359)
- `Robot` (page 179) now has a proper `value` (page 180) attribute (#305)
- Added `CPUtemperature` (page 194) as another demo of “internal” devices (#294)
- A temporary work-around for an issue with `DistanceSensor` (page 115) 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!

26.12 Release 1.2.0 (2016-04-10)

- Added `Energenie` (page 182) class for controlling Energenie plugs (#69)
- Added `LineSensor` (page 110) class for single line-sensors (#109)
- Added `DistanceSensor` (page 115) class for HC-SR04 ultra-sonic sensors (#114)
- Added `SnowPi` (page 185) class for the Ryan-tek Snow-pi board (#130)
- Added `when_held` (page 109) (and related properties) to `Button` (page 107) (#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 243) (#180)

\[\text{https://github.com/gpiozero/gpiozero/issues/340}\]
\[\text{https://github.com/gpiozero/gpiozero/issues/248}\]
\[\text{https://github.com/gpiozero/gpiozero/issues/359}\]
\[\text{https://github.com/gpiozero/gpiozero/issues/305}\]
\[\text{https://github.com/gpiozero/gpiozero/issues/294}\]
\[\text{https://github.com/gpiozero/gpiozero/issues/385}\]
\[\text{https://github.com/gpiozero/gpiozero/issues/320}\]
\[\text{https://github.com/gpiozero/gpiozero/issues/295}\]
\[\text{https://github.com/gpiozero/gpiozero/issues/289}\]
\[\text{https://github.com/gpiozero/gpiozero/issues/69}\]
\[\text{https://github.com/gpiozero/gpiozero/issues/109}\]
\[\text{https://github.com/gpiozero/gpiozero/issues/114}\]
\[\text{https://github.com/gpiozero/gpiozero/issues/130}\]
\[\text{https://github.com/gpiozero/gpiozero/issues/115}\]
\[\text{https://github.com/gpiozero/gpiozero/issues/140}\]
\[\text{https://github.com/gpiozero/gpiozero/issues/162}\]
\[\text{https://github.com/gpiozero/gpiozero/issues/180}\]
Many refinements to the base classes mean more consistency in composite devices and several bugs squashed (\#164\textsuperscript{1192}, \#175\textsuperscript{1193}, \#182\textsuperscript{1194}, \#189\textsuperscript{1195}, \#193\textsuperscript{1196}, \#229\textsuperscript{1197})

GPIO Zero is now aware of what sort of Pi it’s running on via \texttt{pi.info()} (page 221) 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\textsuperscript{1198})

The source/values system was enhanced to ensure normal usage doesn’t stress the CPU and lots of utilities were added (\#181\textsuperscript{1199}, \#251\textsuperscript{1200})

And I’ll just add a note of thanks to the many people in the community who contributed to this release: we’ve had some great PRs, suggestions, and bug reports in this version. Of particular note:

- Schelto van Doorn was instrumental in adding support for numerous ADC chips
- Alex Eames generously donated a RasPiO Analog board which was extremely useful in developing the software SPI interface (and testing the ADC support)
- Andrew Scheller squashed several dozen bugs (usually a day or so after Dave had introduced them ;)

As always, many thanks to the whole community - we look forward to hearing from you more in 1.3!

\section*{26.13 Release 1.1.0 (2016-02-08)}

- Documentation converted to reST and expanded to include generic classes and several more recipes (\#80\textsuperscript{1201}, \#82\textsuperscript{1202}, \#101\textsuperscript{1203}, \#119\textsuperscript{1204}, \#135\textsuperscript{1205}, \#168\textsuperscript{1206})
- New \texttt{CamJamKitRobot} (page 181) class with the pre-defined motor pins for the new CamJam EduKit
- New \texttt{LEDBarGraph} (page 162) class (many thanks to Martin O’Hanlon!) (\#126\textsuperscript{1207}, \#176\textsuperscript{1208})
- New \texttt{Pin} (page 233) implementation abstracts out the concept of a GPIO pin paving the way for alternate library support and IO extenders in future (\#141\textsuperscript{1209})
- New \texttt{LEDBoard.blink()} (page 160) method which works properly even when background is set to \texttt{False} (\#94\textsuperscript{1210}, \#161\textsuperscript{1211})
- New \texttt{RGBLED.blink()} (page 132) method which implements (rudimentary) color fading too! (\#135\textsuperscript{1212}, \#174\textsuperscript{1213})
- New \texttt{initial_value} attribute on \texttt{OutputDevice} (page 146) ensures consistent behaviour on construction (\#118\textsuperscript{1214})
• New `active_high` attribute on `PWMOutputDevice` (page 144) and `RGBLED` (page 131) allows use of common anode devices (#143<sup>1215</sup>, #154<sup>1216</sup>)
• Loads of new ADC chips supported (many thanks to GitHub user pcpoa!) (#150<sup>1217</sup>)

26.14 Release 1.0.0 (2015-11-16)

• Debian packaging added (#44<sup>1218</sup>)
• `PWMLED` (page 129) class added (#58<sup>1219</sup>)
• `TemperatureSensor` removed pending further work (#93<sup>1220</sup>)
• `Buzzer.beep()` (page 134) alias method added (#75<sup>1221</sup>)
• `Motor` (page 137) PWM devices exposed, and `Robot` (page 179) motor devices exposed (#107<sup>1222</sup>)

26.15 Release 0.9.0 (2015-10-25)

Fourth public beta

• Added source and values properties to all relevant classes (#76<sup>1223</sup>)
• Fix names of parameters in `Motor` (page 137) constructor (#79<sup>1224</sup>)
• Added wrappers for LED groups on add-on boards (#81<sup>1225</sup>)

26.16 Release 0.8.0 (2015-10-16)

Third public beta

• Added generic `AnalogInputDevice` (page 155) class along with specific classes for the `MCP3008` (page 151) and `MCP3004` (page 151) (#41<sup>1226</sup>)
• Fixed `DigitalOutputDevice.blink()` (page 144) (#57<sup>1227</sup>)

26.17 Release 0.7.0 (2015-10-09)

Second public beta

26.18 Release 0.6.0 (2015-09-28)

First public beta

1215 https://github.com/gpiozero/gpiozero/issues/143
1216 https://github.com/gpiozero/gpiozero/issues/154
1217 https://github.com/gpiozero/gpiozero/issues/150
1218 https://github.com/gpiozero/gpiozero/issues/44
1219 https://github.com/gpiozero/gpiozero/issues/58
1220 https://github.com/gpiozero/gpiozero/issues/93
1221 https://github.com/gpiozero/gpiozero/issues/75
1222 https://github.com/gpiozero/gpiozero/issues/107
1223 https://github.com/gpiozero/gpiozero/issues/76
1224 https://github.com/gpiozero/gpiozero/issues/81
1225 https://github.com/gpiozero/gpiozero/issues/41
1226 https://github.com/gpiozero/gpiozero/issues/57
1227 https://github.com/gpiozero/gpiozero/issues/57
26.19 Release 0.5.0 (2015-09-24)

26.20 Release 0.4.0 (2015-09-23)

26.21 Release 0.3.0 (2015-09-22)

26.22 Release 0.2.0 (2015-09-21)

Initial release
Copyright © 2015-2020 Ben Nuttall <ben@bennuttall.com> and contributors; see gpiozero (page 1) for current list

SPDX-License-Identifier: BSD-3-Clause

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.
Python Module Index

g

gpiozero, 3
  gpiozero.boards, 159
  gpiozero.devices, 201
  gpiozero.exc, 247
  gpiozero.fonts, 215
  gpiozero.input_devices, 107
  gpiozero.internal_devices, 191
  gpiozero.output_devices, 127
  gpiozero.pins, 227
  gpiozero.pins.data, 221
  gpiozero.pins.lgpio, 242
  gpiozero.pins.local, 241
  gpiozero.pins.mock, 244
  gpiozero.pins.native, 243
  gpiozero.pins.pi, 240
  gpiozero.pins.rpigpio, 242
  gpiozero.pins.rpigpio, 241
  gpiozero.spi_devices, 149
  gpiozero.tones, 219
  gpiozero.tools, 207
Symbols

- **c**, --color
  pinout command line option, 77
- **h**, --help
  pinout command line option, 77
- **m**, --monochrome
  pinout command line option, 77
- **r** REVISION, --revision REVISION
  pinout command line option, 77
- **x**, --xyz
  pinout command line option, 77

_shared_key() (gpiozero.SharedMixin class method), 204

A

absoluted() (in module gpiozero.tools), 207
active_high (gpiozero.OutputDevice attribute), 147
active_time (gpiozero.DigitalInputDevice attribute), 122
active_time (gpiozero.EventsMixin attribute), 205
all_values() (in module gpiozero.tools), 211
alternating_values() (in module gpiozero.tools), 213
amber (gpiozero.TrafficLights attribute), 170
AnalogInputDevice (class in gpiozero), 155
AngularServo (class in gpiozero), 141
any_values() (in module gpiozero.tools), 211
arms (gpiozero.SnowPi attribute), 185
averaged() (in module gpiozero.tools), 211

B

backward() (gpiozero.Motor method), 137
backward() (gpiozero.PhaseEnableMotor method), 138
backward() (gpiozero.Robot method), 179
BadEventHandler, 248
BadPinFactory, 248
BadQueueLen, 248
BadWaitTime, 248
beep() (gpiozero.Buzzer method), 134
bits (gpiozero.AnalogInputDevice attribute), 156
bits_per_word (gpiozero.SPI attribute), 237
blink() (gpiozero.DigitalOutputDevice method), 144
blink() (gpiozero.LED method), 128
blink() (gpiozero.LEDBoard method), 160
blink() (gpiozero.PWMLED method), 129
blink() (gpiozero.PWMOoutputDevice method), 145
blink() (gpiozero.RGBLED method), 132
blue (gpiozero.RGBLED attribute), 133
bluetooth (gpiozero.PiBoardInfo attribute), 224
board (gpiozero.PiBoardInfo attribute), 224
booleanized() (in module gpiozero.tools), 207
bounce (gpiozero.Pin attribute), 234
Button (class in gpiozero), 107
button (gpiozero.Pibrella attribute), 178
button (gpiozero.StatusBoard attribute), 185
button (gpiozero.TrafficLightsBuzzer attribute), 171
ButtonBoard (class in gpiozero), 167
Buzzer (class in gpiozero), 134
buzzer (gpiozero.JamHat attribute), 177
buzzer (gpiozero.Pibrella attribute), 179
buzzer (gpiozero.TrafficLightsBuzzer attribute), 171

C

CallbackSetToNone, 252
CamJamKitRobot (class in gpiozero), 181
channel (gpiozero.MCP3002 attribute), 150
channel (gpiozero.MCP3004 attribute), 151
channel (gpiozero.MCP3008 attribute), 151
channel (gpiozero.MCP3202 attribute), 152
channel (gpiozero.MCP3204 attribute), 153
channel (gpiozero.MCP3208 attribute), 153
channel (gpiozero.MCP3302 attribute), 153
channel (gpiozero.MCP3304 attribute), 154
clamped() (in module gpiozero.tools), 208
clock_mode (gpiozero.SPI attribute), 237
clock_phase (gpiozero.SPI attribute), 237
clock_polarity (gpiozero.SPI attribute), 238
close() (gpiozero.CompositeDevice method), 189
close() (gpiozero.Device method), 203
close() (gpiozero.Factory method), 232

267
close() (gpiozero.GPIODevice method), 125
close() (gpiozero.Pin method), 234
close() (gpiozero.pins.pi.PinFactory method), 240
close() (gpiozero.SPIDevice method), 156
closed (gpiozero.CompositeDevice attribute), 190
closed (gpiozero.Device attribute), 203
closed (gpiozero.GPIODevice attribute), 126
col (gpiozero.PiPinInfo attribute), 125
color (gpiozero.RGBLED attribute), 133
columns (gpiozero.HeaderInfo attribute), 225
CompositeDevice (class in gpiozero), 188
CompositeDeviceBadDevice, 248
CompositeDeviceBadName, 248
CompositeDeviceBadOrder, 248
CompositeDeviceError, 248
CompositeOutputDevice (class in gpiozero), 188
cos_values() (in module gpiozero.tools), 213
CPUPerformance (class in gpiozero), 194
csi (gpiozero.PiBoardInfo attribute), 224

detach() (gpiozero.PiBoardInfo attribute), 140
Device (class in gpiozero), 203
DeviceClosed, 247
differential (gpiozero.MCP3002 attribute), 150
differential (gpiozero.MCP3004 attribute), 151
differential (gpiozero.MCP3008 attribute), 151
differential (gpiozero.MCP3202 attribute), 152
differential (gpiozero.MCP3204 attribute), 152
differential (gpiozero.MCP3208 attribute), 153
differential (gpiozero.MCP3302 attribute), 153
differential (gpiozero.MCP3304 attribute), 154
DigitalInputDevice (class in gpiozero), 121
DigitalOutputDevice (class in gpiozero), 143
DiskUsage (class in gpiozero), 197
distance (gpiozero.DistanceSensor attribute), 117
DistanceSensor (class in gpiozero), 115
DistanceSensorNoEcho, 251
down() (gpiozero.tones.Tone method), 220
daI (gpiozero.PiBoardInfo attribute), 224

E

echo (gpiozero.DistanceSensor attribute), 117
edges (gpiozero.Pin attribute), 235
end_time (gpiozero.TimeOfDay attribute), 193
Energenie (class in gpiozero), 182
EnergenieBadSocket, 248
EnergenieSocketMissing, 248
environment variable

GPIOZERO_PIN_FACTORY, 52, 62, 80, 83, 89, 229, 230
GPIOZERO_TEST_INPUT_PIN, 104
GPIOZERO_TEST_PIN, 104
PIGPIO_ADDA, 52, 62, 80, 230
PIGPIO_PORT, 80
eth_speed (gpiozero.PiBoardInfo attribute), 223
ethernet (gpiozero.PiBoardInfo attribute), 223

EventsMixin (class in gpiozero), 205
eyes (gpiozero.PumpkinPi attribute), 186
eyes (gpiozero.SnowPi attribute), 185

F

Factory (class in gpiozero), 232
FishDish (class in gpiozero), 175
font (gpiozero.LEDCharDisplay attribute), 165
forward() (gpiozero.Motor method), 137
forward() (gpiozero.PhaseEnableMotor method), 139
forward() (gpiozero.Robot method), 180
frame_width (gpiozero.Servo attribute), 140
frequency (gpiozero.Pin attribute), 235
frequency (gpiozero.PWMOutputDevice attribute), 146
frequency (gpiozero.tones.Tone attribute), 220
from_frequency() (gpiozero.tones.Tone class method), 220
from_key() (gpiozero.tones.Tone class method), 220
from_note() (gpiozero.tones.Tone class method), 220
function (gpiozero.Pin attribute), 235
function (gpiozero.PinInfo attribute), 225

G

GPIODevice (class in gpiozero), 125
GPIODeviceClosed, 249
GPIODeviceError, 249
GPIOInUse, 249
GPIOPinMissing, 249
gpiozero (module), 3
gpiozero.boards (module), 159
gpiozero.devices (module), 201
gpiozero.exc (module), 247
gpiozero.fonts (module), 215
gpiozero.input_devices (module), 107
gpiozero.internal_devices (module), 191
gpiozero.output_devices (module), 127
gpiozero.pins (module), 227
gpiozero.pins.data (module), 221
gpiozero.pins.lgpio (module), 242
gpiozero.pins.local (module), 241
gpiozero.pins.mock (module), 244
gpiozero.pins.native (module), 243
gpiozero.pins.pi (module), 240
gpiozero.pins.pigpio (module), 242
gpiozero.pins.rpigpio (module), 241
gpiozero.spi_devices (module), 149
gpiozero.tones (module), 219
gpiozero.tools (module), 207
GPIOZERO_PIN_FACTORY, 52, 62, 83, 89, 229, 230
GPIOZERO_TEST_INPUT_PIN, 104
GPIOZERO_TEST_PIN, 104
GPIOZeroError, 247
GPIOZeroWarning, 251
green (gpiozero.RGBLED attribute), 133

Index
green (gpiozero.StatusBoard attribute), 185
green (gpiozero.StatusZero attribute), 184
green (gpiozero.TrafficLights attribute), 171

H
HeaderInfo (class in gpiozero), 224
headers (gpiozero.PiBoardInfo attribute), 224
held_time (gpiozero.Button attribute), 108
held_time (gpiozero.HoldMixin attribute), 206
hold_repeat (gpiozero.Button attribute), 109
hold_repeat (gpiozero.HoldMixin attribute), 206
hold_time (gpiozero.Button attribute), 109
hold_time (gpiozero.HoldMixin attribute), 206
HoldMixin (class in gpiozero), 206
host (gpiozero.PingServer attribute), 194

I
inactive_time (gpiozero.DigitalInputDevice attribute), 122
 inactive_time (gpiozero.EventsMixin attribute), 205
input_with_pull() (gpiozero.Pin method), 234
InputDevice (class in gpiozero), 124
InputDeviceError, 250
inputs (gpiozero.Pibrella attribute), 179
InternalDevice (class in gpiozero), 199
inverted() (in module gpiozero.tools), 208
is_active (gpiozero.AngularServo attribute), 142
is_active (gpiozero.Buzzer attribute), 135
is_active (gpiozero.CompositeDevice attribute), 190
is_active (gpiozero.CPUTemperature attribute), 195
is_active (gpiozero.Device attribute), 203
is_active (gpiozero.DiskUsage attribute), 198
is_active (gpiozero.InputDevice attribute), 125
is_active (gpiozero.LoadAverage attribute), 197
is_active (gpiozero.Motor attribute), 138
is_active (gpiozero.PhaseEnableMotor attribute), 139
is_active (gpiozero.PingServer attribute), 194
is_active (gpiozero.PWMOutputDevice attribute), 146
is_active (gpiozero.Servo attribute), 140
is_active (gpiozero.SmoothedInputDevice attribute), 124
is_active (gpiozero.TimeOfDay attribute), 193
is_active (gpiozero.TonalBuzzer attribute), 136
is_held (gpiozero.Button attribute), 109
is_held (gpiozero.HoldMixin attribute), 206
is_lit (gpiozero.LEDLED attribute), 128
is_lit (gpiozero.LEDCollection attribute), 187
is_lit (gpiozero.PWMLED attribute), 130
is_lit (gpiozero.RGBLED attribute), 133
is_pressed (gpiozero.Button attribute), 109
is_pressed (gpiozero.ButtonBoard attribute), 169

J
JamHat (class in gpiozero), 177

L
LED (class in gpiozero), 127
LEDBarGraph (class in gpiozero), 162
LEDBoard (class in gpiozero), 159
LedBorg (class in gpiozero), 172
LEDCollection (class in gpiozero), 187
LEDMultiCharDisplay (class in gpiozero), 166
leds (gpiozero.LEDCollection attribute), 187
left() (gpiozero.Robot method), 180
left_motor (gpiozero.Robot attribute), 179
LGPIOFactory (class in gpiozero.pins.lgpio), 242
LGPIOPin (class in gpiozero.pins.lgpio), 242
light_detected (gpiozero.LightSensor attribute), 115
lights (gpiozero.Pibrella attribute), 178
lights (gpiozero.StatusBoard attribute), 184
lights (gpiozero.TrafficLightsBuzzer attribute), 171
LightSensor (class in gpiozero), 113
LineSensor (class in gpiozero), 110
lit_count (gpiozero.LEDBarGraph attribute), 163
load_average (gpiozero.LoadAverage attribute), 197
load_font_14seg() (in module gpiozero.fonts), 216
load_font_7seg() (in module gpiozero.fonts), 215
load_segment_font() (in module gpiozero.fonts), 216
LoadAverage (class in gpiozero), 196
LocalPiFactory (class in gpiozero.pins.local), 241
LocalPiPin (class in gpiozero.pins.local), 241
lsb_first (gpiozero.SPI attribute), 238

M
manufacturer (gpiozero.PiBoardInfo attribute), 223
max() (gpiozero.AngularServo method), 142
max() (gpiozero.Servo method), 140
max_angle (gpiozero.AngularServo attribute), 142
max_distance (gpiozero.DistanceSensor attribute), 117
max_pulse_width (gpiozero.Servo attribute), 140
max_steps (gpiozero.RotaryEncoder attribute), 119
max_tone (gpiozero.TonalBuzzer attribute), 136
max_voltage (gpiozero.AnalogInputDevice attribute), 156
MCP3001 (class in gpiozero), 150
MCP3002 (class in gpiozero), 150
MCP3004 (class in gpiozero), 151
MCP3008 (class in gpiozero), 151
MCP3201 (class in gpiozero), 152
MCP3202 (class in gpiozero), 152
MCP3204 (class in gpiozero), 152
MCP3208 (class in gpiozero), 153
MCP3301 (class in gpiozero), 153
MCP3302 (class in gpiozero), 153
MCP3304 (class in gpiozero), 154
memory (gpiozero.PiBoardInfo attribute), 223
midi() (gpiozero.TonalBuzzer attribute), 136
midi (gpiozero.TonalBuzzer attribute), 136
min() (gpiozero.TonalBuzzer attribute), 136
min (gpiozero.TonalBuzzer attribute), 136
MockChargingPin (class in gpiozero.pins.mock), 244
MockConnectedPin (class in gpiozero.pins.mock), 244
MockFactory (class in gpiozero.pins.mock), 244
MockPWMPin (class in gpiozero.pins.mock), 244
MockSPIDevice (class in gpiozero.pins.mock), 245
MockTriggerPin (class in gpiozero.pins.mock), 244
model (gpiozero.PiBoardInfo attribute), 223
motion_detected (gpiozero.MotionSensor attribute), 113
MotionSensor (class in gpiozero), 112
Motor (class in gpiozero), 137
multiplied() (in module gpiozero.tools), 212

N
ame (gpiozero.HeaderInfo attribute), 225
namedtuple (gpiozero.CompositeDevice attribute), 190
Native2711Pin (class in gpiozero.pins.native), 243
Native2835Pin (class in gpiozero.pins.native), 243
NativeFactory (class in gpiozero.pins.native), 243
NativePin (class in gpiozero.pins.native), 243
negated() (in module gpiozero.tools), 208
nose (gpiozero.SnowPi attribute), 185
note (gpiozero.Tones.Tone attribute), 220
number (gpiozero.PinInfo attribute), 225

O
octaves (gpiozero.TonalBuzzer attribute), 136
off() (gpiozero.Buzzer method), 135
off() (gpiozero.CompositeOutputDevice method), 188
off() (gpiozero.DigitalOutputDevice method), 144
off() (gpiozero.Energenie method), 183
off() (gpiozero.JamHat method), 178
off() (gpiozero.LED method), 128
off() (gpiozero.LEDBoard method), 161
off() (gpiozero.OutputDevice method), 147
off() (gpiozero.Pibrella method), 179
off() (gpiozero.PWMDevice method), 130
off() (gpiozero.PWMOutputDevice method), 145
off() (gpiozero.RGBLED method), 132
on() (gpiozero.Buzzer method), 135
on() (gpiozero.CompositeOutputDevice method), 188
on() (gpiozero.DigitalOutputDevice method), 144
on() (gpiozero.Energenie method), 183
on() (gpiozero.JamHat method), 178
on() (gpiozero.LED method), 128
on() (gpiozero.LEDBoard method), 161
on() (gpiozero.OutputDevice method), 147
on() (gpiozero.Pibrella method), 179
on() (gpiozero.PWMDevice method), 130
on() (gpiozero.PWMOutputDevice method), 145
on() (gpiozero.RGBLED method), 132

P
partial (gpiozero.SmoothedInputDevice attribute), 124
pcb_revision (gpiozero.PiBoardInfo attribute), 223
PhaseEnableMotor (class in gpiozero), 138
physical_pin() (gpiozero.PiBoardInfo method), 222
physical_pins() (gpiozero.PiBoardInfo method), 222
pi_info (gpiozero.Factory attribute), 233
pi_info() (in module gpiozero), 221
PiBoardInfo (class in gpiozero), 221
Pibrella (class in gpiozero), 178
PiFactory (class in gpiozero.pins.pi), 240
PIGPIO_ADDR, 52, 62, 230
PiGPIOFactory (class in gpiozero.pins.pigpio), 242
PiGPIOPin (class in gpiozero.pins.pigpio), 243
PiHutXmasTree (class in gpiozero), 171
PiLight (class in gpiozero), 173
PiLightBarGraph (class in gpiozero), 174
Pin (class in gpiozero), 233
pin (gpiozero.Button attribute), 109
pin (gpiozero.Buzzer attribute), 135
pin (gpiozero.GPIODevice attribute), 126
pin (gpiozero.LED attribute), 128
pin (gpiozero.LightSensor attribute), 115
pin (gpiozero.LineSensor attribute), 111
pin (gpiozero.MotionSensor attribute), 113
pin (gpiozero.PWMDevice attribute), 130
pin() (gpiozero.Factory method), 232
pin() (gpiozero.pins.mock.MockFactory method), 244
pin() (gpiozero.pins.pi.PiFactory method), 240
pin_class (gpiozero.pins.mock.MockFactory attribute), 244
PinFactory (gpiozero.Device attribute), 203
PinEdgeDetectUnsupported, 250
PinError, 250
Index
socket (gpiozero.Energenie attribute), 183
source (gpiozero.LEDBarGraph attribute), 163
source (gpiozero.SourceMixin attribute), 204
source_delay (gpiozero.SourceMixin attribute), 204
SourceMixin (class in gpiozero), 204
SPI (class in gpiozero), 236
spi() (gpiozero.Factory method), 233
spi() (gpiozero.pins.pi.PiFactory method), 240
SPIBadArgs, 249
SPIBadChannel, 249
SPIDevice (class in gpiozero), 156
SPIError, 248
SPIFixedBitOrder, 249
SPIFixedClockMode, 249
SPIFixedSelect, 249
SPIFixedWordSize, 249
SPIInvalidClockMode, 147
SPIInvalidWordSize, 147
toggle() (gpiozero.CompositeOutputDevice method), 188
toggle() (gpiozero.LED method), 128
toggle() (gpiozero.LEDBoard method), 162
toggle() (gpiozero.OutputDevice method), 147
toggle() (gpiozero.PWMBoard method), 130
toggle() (gpiozero.PWMOOutputDevice method), 146
toggle() (gpiozero.RGBLED method), 133
TonalBuzzer (class in gpiozero), 135
Tone (class in gpiozero.tones), 219
tone (gpiozero.TonalBuzzer attribute), 136
TrafficHat (class in gpiozero), 176
TrafficLights (class in gpiozero), 170
TrafficLightsBuzzer (class in gpiozero), 171
TrafficPcap (class in gpiozero), 176
transfer() (gpiozero.SPI method), 236
trigger (gpiozero.DistanceSensor attribute), 117
U
up() (gpiozero.tones.Tone method), 220
usage (gpiozero.DiskUsage attribute), 198
usb (gpiozero.PiBoardInfo attribute), 223
usb3 (gpiozero.PiBoardInfo attribute), 223
utc (gpiozero.TimeOfDay attribute), 193
V
value (gpiozero.AnalogInputDevice attribute), 156
value (gpiozero.AngularServo attribute), 143
value (gpiozero.Button attribute), 109
value (gpiozero.ButtonBoard attribute), 169
value (gpiozero.Buzzer attribute), 135
value (gpiozero.CompositeDevice attribute), 190
value (gpiozero.CompositeOutputDevice attribute), 188
value (gpiozero.CPUTemperature attribute), 195
value (gpiozero.Device attribute), 203
value (gpiozero.DigitalInputDevice attribute), 122
value (gpiozero.DigitalOutputDevice attribute), 144
value (gpiozero.DiskUsage attribute), 198
value (gpiozero.DistanceSensor attribute), 117
value (gpiozero.Encrygenie attribute), 183
value (gpiozero.GPIODevice attribute), 126
value (gpiozero.InputDevice attribute), 125
value (gpiozero.LED attribute), 129
value (gpiozero.LEDBarGraph attribute), 163
value (gpiozero.LEDCharDisplay attribute), 165
value (gpiozero.LEDMultiCharDisplay attribute), 166
value (gpiozero.LightSensor attribute), 115
value (gpiozero.LineSensor attribute), 111
value (gpiozero.LoadAverage attribute), 197
value (gpiozero.MCP3001 attribute), 150
value (gpiozero.MCP3002 attribute), 151
value (gpiozero.MCP3004 attribute), 151
value (gpiozero.MCP3008 attribute), 151
value (gpiozero.MCP3201 attribute), 152

T

temperature (gpiozero.CPUTemperature attribute), 195
threshold (gpiozero.SmoothedInputDevice attribute), 124
threshold_distance (gpiozero.DistanceSensor attribute), 117
threshold_steps (gpiozero.RotaryEncoder attribute), 119
stop() (gpiozero.Motor method), 138
stop() (gpiozero.PhaseEnableMotor method), 139
stop() (gpiozero.Robot method), 180
storage (gpiozero.PiBoardInfo attribute), 223
summed() (in module gpiozero.tools), 212

U
up() (gpiozero.tones.Tone method), 220
usage (gpiozero.DiskUsage attribute), 198
usb (gpiozero.PiBoardInfo attribute), 223
usb3 (gpiozero.PiBoardInfo attribute), 223
utc (gpiozero.TimeOfDay attribute), 193

V
value (gpiozero.AnalogInputDevice attribute), 156
value (gpiozero.AngularServo attribute), 143
value (gpiozero.Button attribute), 109
value (gpiozero.ButtonBoard attribute), 169
value (gpiozero.Buzzer attribute), 135
value (gpiozero.CompositeDevice attribute), 190
value (gpiozero.CompositeOutputDevice attribute), 188
value (gpiozero.CPUTemperature attribute), 195
value (gpiozero.Device attribute), 203
value (gpiozero.DigitalInputDevice attribute), 122
value (gpiozero.DigitalOutputDevice attribute), 144
value (gpiozero.DiskUsage attribute), 198
value (gpiozero.DistanceSensor attribute), 117
value (gpiozero.Encrygenie attribute), 183
value (gpiozero.GPIODevice attribute), 126
value (gpiozero.InputDevice attribute), 125
value (gpiozero.LED attribute), 129
value (gpiozero.LEDBarGraph attribute), 163
value (gpiozero.LEDCharDisplay attribute), 165
value (gpiozero.LEDMultiCharDisplay attribute), 166
value (gpiozero.LightSensor attribute), 115
value (gpiozero.LineSensor attribute), 111
value (gpiozero.LoadAverage attribute), 197
value (gpiozero.MCP3001 attribute), 150
value (gpiozero.MCP3002 attribute), 151
value (gpiozero.MCP3004 attribute), 151
value (gpiozero.MCP3008 attribute), 151
value (gpiozero.MCP3201 attribute), 152

T

temperature (gpiozero.CPUTemperature attribute), 195
threshold (gpiozero.SmoothedInputDevice attribute), 124
threshold_distance (gpiozero.DistanceSensor attribute), 117
threshold_steps (gpiozero.RotaryEncoder attribute), 119
stop() (gpiozero.Motor method), 138
stop() (gpiozero.PhaseEnableMotor method), 139
stop() (gpiozero.Robot method), 180
storage (gpiozero.PiBoardInfo attribute), 223
summed() (in module gpiozero.tools), 212

U
up() (gpiozero.tones.Tone method), 220
usage (gpiozero.DiskUsage attribute), 198
usb (gpiozero.PiBoardInfo attribute), 223
usb3 (gpiozero.PiBoardInfo attribute), 223
utc (gpiozero.TimeOfDay attribute), 193

V
value (gpiozero.AnalogInputDevice attribute), 156
value (gpiozero.AngularServo attribute), 143
value (gpiozero.Button attribute), 109
value (gpiozero.ButtonBoard attribute), 169
value (gpiozero.Buzzer attribute), 135
value (gpiozero.CompositeDevice attribute), 190
value (gpiozero.CompositeOutputDevice attribute), 188
value (gpiozero.CPUTemperature attribute), 195
value (gpiozero.Device attribute), 203
value (gpiozero.DigitalInputDevice attribute), 122
value (gpiozero.DigitalOutputDevice attribute), 144
value (gpiozero.DiskUsage attribute), 198
value (gpiozero.DistanceSensor attribute), 117
value (gpiozero.Encrygenie attribute), 183
value (gpiozero.GPIODevice attribute), 126
value (gpiozero.InputDevice attribute), 125
value (gpiozero.LED attribute), 129
value (gpiozero.LEDBarGraph attribute), 163
value (gpiozero.LEDCharDisplay attribute), 165
value (gpiozero.LEDMultiCharDisplay attribute), 166
value (gpiozero.LightSensor attribute), 115
value (gpiozero.LineSensor attribute), 111
value (gpiozero.LoadAverage attribute), 197
value (gpiozero.MCP3001 attribute), 150
value (gpiozero.MCP3002 attribute), 151
value (gpiozero.MCP3004 attribute), 151
value (gpiozero.MCP3008 attribute), 151
value (gpiozero.MCP3201 attribute), 152
value (gpiozero.MCP3202 attribute), 152
value (gpiozero.MCP3204 attribute), 152
value (gpiozero.MCP3208 attribute), 153
value (gpiozero.MCP3301 attribute), 153
value (gpiozero.MCP3302 attribute), 153
value (gpiozero.MCP3304 attribute), 154
value (gpiozero.MotionSensor attribute), 113
value (gpiozero.Motor attribute), 138
value (gpiozero.OutputDevice attribute), 147
value (gpiozero.PhaseEnableMotor attribute), 139
value (gpiozero.PingServer attribute), 194
value (gpiozero.PWMLED attribute), 130
value (gpiozero.PWMOutputDevice attribute), 146
value (gpiozero.RGBLED attribute), 133
value (gpiozero.Robot attribute), 180
value (gpiozero.RotaryEncoder attribute), 120
value (gpiozero.Servo attribute), 141
value (gpiozero.SmoothedInputDevice attribute), 124
value (gpiozero.TimeOfDay attribute), 193
value (gpiozero.TonalBuzzer attribute), 136
values (gpiozero.LEDBarGraph attribute), 164
values (gpiozero.ValuesMixin attribute), 204
ValuesMixin (class in gpiozero), 204
voltage (gpiozero.AnalogInputDevice attribute), 156

W

wait_for_active() (gpiozero.ButtonBoard method), 122
wait_for_active() (gpiozero.ButtonBoard method), 205
wait_for_dark() (gpiozero.LightSensor method), 114
wait_for_in_range() (gpiozero.DistanceSensor method), 117
wait_for_inactive() (gpiozero.ButtonBoard method), 122
wait_for_inactive() (gpiozero.ButtonBoard method), 205
wait_for_light() (gpiozero.LightSensor method), 114
wait_for_line() (gpiozero.LineSensor method), 111
wait_for_motion() (gpiozero.MotionSensor method), 113
wait_for_no_line() (gpiozero.LineSensor method), 111
wait_for_no_motion() (gpiozero.MotionSensor method), 113
wait_for_out_of_range() (gpiozero.DistanceSensor method), 117
wait_for_press() (gpiozero.Button method), 108
wait_for_press() (gpiozero.ButtonBoard method), 169
wait_for_release() (gpiozero.Button method), 108

wait_for_release() (gpiozero.ButtonBoard method), 169
wait_for_rotate() (gpiozero.RotaryEncoder method), 119
wait_for_rotate_clockwise() (gpiozero.RotaryEncoder method), 119
wait_for_rotate_counter_clockwise() (gpiozero.RotaryEncoder method), 119
when_activated (gpiozero.CPUTemperature attribute), 195
when_activated (gpiozero.DigitalInputDevice attribute), 122
when_activated (gpiozero.DiskUsage attribute), 198
when_activated (gpiozero.EventsMixin attribute), 205
when_activated (gpiozero.LoadAverage attribute), 197
when_activated (gpiozero.PingServer attribute), 194
when_activated (gpiozero.TimeOfDay attribute), 193
when_changed (gpiozero.Pin attribute), 236
when_dark (gpiozero.LightSensor attribute), 115
when_deactivated (gpiozero.CPUTemperature attribute), 196
when_deactivated (gpiozero.DigitalInputDevice attribute), 122
when_deactivated (gpiozero.DiskUsage attribute), 198
when_deactivated (gpiozero.EventsMixin attribute), 205
when_deactivated (gpiozero.LoadAverage attribute), 197
when_deactivated (gpiozero.PingServer attribute), 194
when_deactivated (gpiozero.TimeOfDay attribute), 193
when_held (gpiozero.Button attribute), 109
when_held (gpiozero.HoldMixin attribute), 206
when_in_range (gpiozero.DistanceSensor attribute), 117
when_light (gpiozero.LightSensor attribute), 115
when_line (gpiozero.LineSensor attribute), 111
when_motion (gpiozero.MotionSensor attribute), 113
when_no_line (gpiozero.LineSensor attribute), 111
when_no_motion (gpiozero.MotionSensor attribute), 113
when_out_of_range (gpiozero.DistanceSensor attribute), 118
when_pressed (gpiozero.Button attribute), 109
when_pressed (gpiozero.ButtonBoard attribute), 169
when_released (gpiozero.Button attribute), 109
when_released (gpiozero.ButtonBoard attribute), 169
when_rotated (gpiozero.RotaryEncoder attribute),
when_rotated_clockwise (gpiozero.RotaryEncoder attribute), 120
when_rotated_counter_clockwise (gpiozero.RotaryEncoder attribute), 120
wifi (gpiozero.PiBoardInfo attribute), 223
wrap (gpiozero.RotaryEncoder attribute), 120
write() (gpiozero.SPI method), 236

Y
yellow (gpiozero.TrafficLights attribute), 171

Z
zip_values() (in module gpiozero.tools), 212
ZombieThread, 248