gpiozero 2.0 Documentation

Release 2.0

Ben Nuttall

Oct 01, 2023
CHAPTER ONE

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\(^4\) 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 45) 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
```

1.1.2 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\(^8\) 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 101) page.

---

1 https://www.raspberrypi.org/software/operating-systems/
2 https://www.raspberrypi.org/software/operating-systems/
5 https://packages.debian.org/buster/python3-gpiozero
6 https://packages.ubuntu.com/hirsute/python3-gpiozero
7 https://pypi.org/project/gpiozero/
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 45) 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<sup>9</sup> 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<sup>10</sup>. 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.

---

<sup>9</sup> https://wiki.debian.org/doc-base

<sup>10</sup> 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\(^2\) 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")
>>> led = LED("WPI0")
```

\(^2\)https://projects.drogon.net/raspberry-pi/wiringpi/pins/
Note that these alternate schemes are merely translations. If you request the state of a device on the command line, the associated pin number will always be reported in the Broadcom (BCM) scheme:

```python
>>> led = LED("J8:11")

Notethatthesesalternateschemesaremerelytranslations. Ifyourequestthestateofadeviceonthecommandline,
theassociatedpinnumberwillalwaysbereportedintheBroadcom(BCM)scheme:

>>> 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 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

red = LED(17)

red.blink()
```

(continues on next page)
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()`\(^{13}\). See *How do I keep my script running?* (page 79) 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 127) using values between 0 and 1:

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

led = PWMLED(17)

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

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

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

led = PWMLED(17)
```

\(^{13}\) [https://docs.python.org/3.9/library/signal.html#signal.pause](https://docs.python.org/3.9/library/signal.html#signal.pause)
led.pulse()
pause()

2.5 Button

Check if a Button (page 105) 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!")
```
**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`\(^\text{(14)}\) (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()
```

### 2.6 Button controlled LED

![Diagram of button controlling LED](image)

\(^\text{(14)}\) `https://docs.python.org/3.9/library/constants.html#None`
Turn on an LED (page 125) when a Button (page 105) 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 `camera.capture()` method requires an output parameter. However, this can be achieved using a custom function which requires no parameters:

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

button = Button(2)
camera = PiCamera()

def capture():
    camera.capture(f'/home/pi/{datetime.now():%Y-%m-%d-%H-%M-%S}.jpg')

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():
    camera.capture(f'/home/pi/{datetime.now():%Y-%m-%d-%H-%M-%S}.jpg')
```

(continues on next page)
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 105) 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()
```

2.9 LEDBoard

A collection of LEDs can be accessed using `LEDBoard` (page 157):

```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 157) 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 157) examples in the advanced **LEDBoard recipes** (page 37).

## 2.10 LEDBarGraph

A collection of LEDs can be treated like a bar graph using **LEDBarGraph** (page 160):

```python
from gpiozero import LEDBarGraph
from time import sleep

graph = LEDBarGraph(5, 6, 13, 19, 26)
graph.value = 1  # (1, 1, 1, 1, 1)
sleep(1)
```

(continues on next page)
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)

Graph values are essentially rounded to account for the fact that LEDs can only be on or off when \( \text{pwm}=\text{False} \) (the default).

However, using \texttt{LEDBarGraph} (page 160) with \( \text{pwm}=\text{True} \) allows more precise values using LED brightness:

```python
from gpiozero import LEDBarGraph
from time import sleep

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

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

graph.value = 3/10  # (1, 0.5, 0, 0, 0)
sleep(1)

graph.value = -3/10 # (0, 0, 0, 0.5, 1)
sleep(1)

graph.value = 9/10  # (1, 1, 1, 1, 0.5)
sleep(1)

graph.value = 95/100 # (1, 1, 1, 1, 0.75)
sleep(1)
```

2.11 LEDCharDisplay

A common 7-segment display\(^{15}\) can be used to represent a variety of characters using \texttt{LEDCharDisplay} (page 162) (which actually supports an arbitrary number of segments):

\(^{15}\) https://en.wikipedia.org/wiki/Seven-segment_display
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 = '321GO'

pause()

See a multi-character example in the advanced recipes (page 38) chapter.

## 2.12 Traffic Lights

A full traffic lights system.

Using a TrafficLights (page 167) kit like Pi-Stop:

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

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

Alternatively:

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

lights = TrafficLights(2, 3, 4)

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

lights.source = traffic_light_sequence()

pause()
```

Using **LED** (page 125) 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()
```
```python
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(f'/home/pi/frame/frame:{frame}.jpg')
    frame += 1
```

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

2.14 Reaction Game

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

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

led = LED(17)
player_1 = Button(2)
player_2 = Button(3)
time = random.uniform(5, 10)
sleep(time)
led.on()

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

led.off()

See Quick Reaction Game\(^7\) for a full resource.

## 2.15 GPIO Music Box

Each button plays a different sound!

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

\(^7\) https://projects.raspberry.pi.org/en/projects/python-quick-reaction-game
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 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 173):

```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 173):

```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 125), **Buzzer** (page 131), and **Button** (page 105) 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 128):

```
from gpiozero import RGBLED
from time import sleep

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

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

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

(continues on next page)
2.18 Motion sensor

Light an LED (page 125) when a MotionSensor (page 109) detects motion:

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

pir = MotionSensor(4)
led = LED(16)
```
2.19 Light sensor

Have a `LightSensor` detect light and dark:

```python
from gpiozero import LightSensor

sensor = LightSensor(18)

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

Run a function when the light changes:

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

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

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

pause()
```

Or make a `PWMLED` change brightness according to the detected light level:
from gpiozero import LightSensor, PWMLED
from signal import pause

sensor = LightSensor(18)
led = PWMLED(16)
led.source = sensor
pause()

2.20 Distance sensor

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

Have a DistanceSensor (page 113) detect the distance to the nearest object:

from gpiozero import DistanceSensor
from time import sleep

sensor = DistanceSensor(23, 24)

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

Run a function when something gets near the sensor:

from gpiozero import DistanceSensor, LED
from signal import pause

(continues on next page)
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 encoder¹⁹ instead.

Have a *RotaryEncoder* (page 115), an *RGBLED* (page 128), and *Button* (page 105) act as a color picker:

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

¹⁹ https://shop.pimoroni.com/products/rotary-encoder-illuminated-rgb
```python
# 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(f'Hue {led.color.hue:.1f}° = {led.color.html}')

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` (page 137) between its minimum, mid-point and maximum positions in sequence:

```python
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` (page 137) between its minimum and maximum positions:

```python
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` (page 137) 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()
```

(continues on next page)
servo.source_delay = 0.1
pause()

Use `AngularServo` (page 139) 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)
```

### 2.23 Motors

Spin a `Motor` (page 134) 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 176) 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 (page 176) with a DistanceSensor (page 113) 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 (page 176):

```python
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 (page 176):

```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
    while True:
        curses.halfdelay(1)
```

(continues on next page)
if next_key is None:
    key = window.getch()
else:
    key = next_key
next_key = None
if key != -1:
    # KEY PRESSED
curses.halfdelay(3)
    action = actions.get(key)
    if action is not None:
        action()
        next_key = key
    while next_key == key:
        next_key = window.getch()
    # KEY RELEASED
    robot.stop()

curses.wrapper(main)

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

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

```python
from gpiozero import Robot, 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
    if set(dev.capabilities().get(ecodes.EV_KEY, [])).isdisjoint(must_not_have)
]
# Pick the first keyboard
keyboard = devices[0]

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

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

20 https://docs.python.org/3.9/library/curses.html#module-curses
if event.value == 0:  # key released
    robot.stop()

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

### 2.27 Motion sensor robot

![Diagram of a motion sensor robot](image)

Make a robot drive forward when it detects motion:
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 149) analog to digital converter:
from gpiozero import MCP3008

pot = MCP3008(channel=0)

while True:
    print(pot.value)

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

from gpiozero import LEDBarGraph, MCP3008
from signal import pause

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

graph.source = pot

pause()

2.29 Measure temperature with an ADC

Wire a TMP36 temperature sensor to the first channel of an MCP3008 (page 149) analog to digital converter:

from gpiozero import MCP3008
from time import sleep

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

adc = MCP3008(channel=0)

for temp in convert_temp(adc.values):
    print('The temperature is', temp, 'C')
sleep(1)
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 200) property rather than a `while` loop:

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

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

(continues on next page)

21 https://docs.python.org/3.9/reference/compound_stmts.html#while
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\(^{22}\) to remotely control the lamp, and the `TimeOfDay` (page 190) 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 191) 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()
```

\(^{22}\) https://energenie4u.co.uk/catalogue/product/ENER002-2PI
2.33 CPU Temperature Bar Graph

You can read the Raspberry Pi's own CPU temperature using the built-in `CPUTemperature` (page 192) class, and display this on a “bar graph” of LEDs:

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

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

leds.source = cpu
pause()
```

2.34 More recipes

Continue to:

- *Advanced Recipes* (page 37)
- *Remote GPIO Recipes* (page 53)
The following recipes demonstrate some of the capabilities of the GPIO Zero library. Please note that all recipes are written assuming Python 3. Recipes may work under Python 2, but no guarantees!

### 3.1 LEDBoard

You can iterate over the LEDs in a `LEDBoard` (page 157) object one-by-one:

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

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

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

`LEDBoard` (page 157) also supports indexing. This means you can access the individual LED (page 125) objects using `leds[i]` where `i` is an integer from 0 up to (not including) the number of LEDs:

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

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

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

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

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

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()
```

(continues on next page)
LEDBoard (page 157) objects can have their LED objects named upon construction. This means the individual LEDs can be accessed by their name:

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

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

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

LEDBoard (page 157) objects can also be nested within other LEDBoard (page 157) objects:

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

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

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

### 3.2 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):
    # (continues on next page)
```
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 2ER0 ')
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),
    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)
```

[23] https://thepihut.com/status
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'

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 travisy. Install with **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 Turtler 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),
    })
```

---

24 https://travispy.readthedocs.io/
{\[btn\])

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()

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()`\(^{25}\) doesn’t support lazy evaluation so the script will simply hang.

\(^{25}\)https://docs.python.org/3.9/library/functions.html#zip
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()
```

Note this recipe requires bluedot and the associated Android app. See the BlueDot documentation\(^\text{26}\) 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
def pos_to_values(x, y):
    left = y if x > 0 else y + x
    right = y if x < 0 else y - x
    return (clamped(left), clamped(right))
```

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

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

robot = Robot(left=Motor(4, 14), right=Motor(17, 18))
bd = BlueDot()
robot.source = drive()
pause()

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:

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

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

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

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

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

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

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

On the Pi 3 the LEDs are controlled by a GPIO expander which is not accessible from gpiozero (yet).
GPIO Zero supports a number of different pin implementations (low-level pin libraries which deal with the GPIO pins directly). By default, the RPi.GPIO\textsuperscript{27} 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 221) documentation page.

One of the pin libraries supported, pigpio\textsuperscript{28}, 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 53) 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\textsuperscript{29}.

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

\textsuperscript{27} https://pypi.python.org/pypi/RPi.GPIO
\textsuperscript{28} http://abyz.me.uk/rpi/pigpio/python.html
\textsuperscript{29} http://abyz.me.uk/rpi/pigpio/download.html
4.1.1 Enable remote connections

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

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
```


\(^{31}\) http://abyz.me.uk/rpi/pigpio/python.html
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-pip32.)

Next, install GPIO Zero and pigpio for Python 3:

$ sudo pip3 install gpiozero pigpio

or Python 2:

$ sudo pip install gpiozero pigpio

4.2.3 Mac OS

First, install pip. If you installed Python 3 using brew, you will already have pip. If not, install pip with get-pip33.

Next, install GPIO Zero and pigpio with pip:

$ pip3 install gpiozero pigpio

Or for Python 2:

$ pip install gpiozero pigpio

4.2.4 Windows

Modern Python installers for Windows bundle pip with Python. If pip is not installed, you can follow this guide34.

Next, install GPIO Zero and pigpio with pip:

C:Usersuser1> pip install gpiozero pigpio

4.3 Environment variables

The simplest way to use devices with remote pins is to set the PIGPIO_ADDR (page 77) environment variable to the IP address of the desired Raspberry Pi. You must run your Python script or launch your development environment with the environment variable set using the command line. For example, one of the following:

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

If you are running this from a PC (not a Raspberry Pi) with gpiozero and the pigpio\(^{35}\) 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 \texttt{PiGPIOFactory} (page 236). If RPi.GPIO\(^{36}\) is installed, this will be selected as the default pin factory, so either uninstall it, or use the \texttt{GPIOZERO_PIN_FACTORY} (page 77) environment variable to override it:

\begin{verbatim}
$ GPIOZERO_PIN_FACTORY=pigpio PIGPIO_ADDR=192.168.1.3 python3 hello.py
\end{verbatim}

This usage will set the pin factory to \texttt{PiGPIOFactory} (page 236) 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:

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

red = LED(17)

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

When run with:

\begin{verbatim}
$ PIGPIO_ADDR=192.168.1.3 python3 led.py
\end{verbatim}

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

\begin{verbatim}
$ PIGPIO_ADDR=192.168.1.4 python3 led.py
\end{verbatim}

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.

\textbf{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 \texttt{PiGPIOFactory} (page 236) can be used, as pigpio\(^{37}\) 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 \texttt{PiGPIOFactory} (page 236) 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:
    (continues on next page)
```

\(^{35}\) http://abyz.me.uk/rpi/pigpio/python.html
\(^{36}\) https://pypi.python.org/pypi/RPi.GPIO
\(^{37}\) http://abyz.me.uk/rpi/pigpio/python.html
led.on()
sleep(1)
led.off()
sleep(1)

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

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

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

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()
```

(continues on next page)
Of course, multiple IP addresses can be used:

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

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

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

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

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

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

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

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

remote_factory = PiGPIOFactory(host='192.168.1.3')

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

You could even use a HAT which is not supported by GPIO Zero (such as the Sense HAT\(^\text{38}\)) on one Pi, and use remote pins to control another over the network:

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

---

\(^{38}\) [https://www.raspberry.org/products/sense-hat/](https://www.raspberry.org/products/sense-hat/)
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 53)
- *Pi Zero USB OTG* (page 57)
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 45).

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 105) on one Raspberry Pi control the LED (page 125) 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 125) 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

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 109) in each room of your house, and have an class:LED indicator showing when there’s motion in each room:

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

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 131) attached in each room you want to hear the doorbell, and use a push Button (page 105) as the doorbell:

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

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

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

for buzzer in buzzers:
    buzzer.source = button

pause()
```

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

5.5 Remote button robot

Similarly to the simple recipe for the button controlled Robot (page 176), 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
```

(continues on next page)
# 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\(^{\text{39}}\) (not supported by GPIO Zero) includes temperature, humidity and pressure sensors, but no light sensor. Remote GPIO allows an external LightSensor (page 111) to be used as well. The Sense HAT LED display can be used to show different colours according to the light levels:

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

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

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

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

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

\(^{39}\) https://www.raspberry.org/products/sense-hat/
The Raspberry PiZero\textsuperscript{40} and Pi Zero W\textsuperscript{41} 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 45) 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\textsuperscript{42} 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 \texttt{usbbootgui} package:

   $ sudo apt update
   $ sudo apt install usbbootgui

\textsuperscript{40} https://www.raspberrypi.org/products/raspberry-pi-zero/
\textsuperscript{41} https://www.raspberrypi.org/products/raspberry-pi-zero-w/
\textsuperscript{42} https://www.raspberrypi.org/downloads/raspberry-pi-desktop/
6.1.3 Ubuntu setup

1. Add the Raspberry Pi PPA to your system:

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

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

   $ sudo pip3 uninstall gpiozero pigpio

3. Install the required packages from the PPA:

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

6.1.4 Access the GPIOs

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

![Raspberry Pi connected dialog]

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%` followed by the connection string. You can look this up using `ifconfig`.

Set the `GPIOZERO_PIN_FACTORY` (page 77) and `PIGPIO_ADDR` (page 77) environment variables on your PC so GPIO Zero connects to the “remote” Pi Zero:

   $ export GPIOZERO_PIN_FACTORY=pigpio
   $ export PIGPIO_ADDR=fe80::1%usb0

Now any GPIO Zero code you run on the PC will use the GPIOs of the attached Pi Zero:
Alternatively, you can set the pin factory in-line, as explained in Configuring Remote GPIO (page 45). Read more on the GPIO expander in blog posts on raspberrypi.org\(^\text{43}\) and bennuttall.com\(^\text{44}\).

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

(See guides on blog.gbaman.info\(^\text{45}\) and learn.adafruit.com\(^\text{46}\) 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 `raspberrypi.local`. SSH into the Pi Zero, install pigpio and run the pigpiodaemon.

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

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

---

\(^\text{43}\) https://www.raspberrypi.org/blog/gpio-expander/
\(^\text{44}\) http://bennuttall.com/raspberry-pi-zero-gpio-expander/
\(^\text{45}\) http://blog.gbaman.info/?p=791
\(^\text{46}\) https://learn.adafruit.com/turning-your-raspberry-pi-zero-into-a-usb-gadget/ethernet-gadget
GPIO Zero provides a method of using the declarative programming paradigm to connect devices together: feeding
the values of one device into another, for example the values of a button into an LED:

\[
\text{Button} \quad \rightarrow \quad \text{LED}
\]

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

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

led.source = button

pause()
```

which is equivalent to:

```python
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 199) 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:

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

Every device also has a `values` (page 200) property (a generator\(^{47}\) continuously yielding the device’s current value). All output devices have a `source` (page 200) 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 200) property (the default is 0.01 seconds).

\(^{47}\) https://wiki.python.org/moin/Generators
\(^{48}\) https://wiki.python.org/moin/Iterator
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:

```
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` property is used to feed values into the output device. Prior to v1.5, the `source` property had to be set directly to a device’s `values` property:

```
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` to a device object.

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

```
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 200):

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:

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 200) 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):

[^59]: https://docs.python.org/3.9/library/constants.html#None
```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 203).

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

![Output device source tool](image)

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

![PWM LED random_values](image)

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

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

Note that in the above example, `source_delay` (page 200) is used to make the LED iterate over the random values slightly slower. `source_delay` (page 200) 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:

![Output device source tool Input device](image)

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

![LED negated Button](image)

```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 200) derivatives, or iterators, including a device's `values` (page 200) property.

Some tools combine the values of multiple sources:

```
Output device

source tool

Input device 1

Input device 2
```

In this example, the LED is lit only if both buttons are pressed (like an AND\(^50\) gate):

```
LED

all_values

Button A

Button B
```

```
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 207) with two buttons would simulate an OR\(^51\) gate.

While most devices have a `value` (page 199) range between 0 and 1, some have a range between -1 and 1 (e.g. `Motor` (page 134), `Servo` (page 137) and `TonalBuzzer` (page 133)). Some source tools output values between -1 and 1, which are ideal for these devices, for example passing `sin_values()` (page 209) in:

---


In this example, all three devices are following the sine wave\footnote{https://en.wikipedia.org/wiki/Sine_wave}. The motor value ramps up from 0 (stopped) to 1 (full speed forwards), then back down to 0 and on to -1 (full speed backwards) in a cycle. Similarly, the servo moves from its midpoint 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 \texttt{source\_delay} (page 200) will alter the speed at which the device iterates through the values. Alternatively, the tool \texttt{cos\_values()} (page 208) could be used to start from -1 and go up to 1, and so on.

### 7.3 Internal devices

GPIO Zero also provides several \textit{internal devices} (page 189) 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 \texttt{values} (page 200) property which can be used to feed values into a device’s \texttt{source} (page 200). For example, a lamp connected to an \texttt{Energenie} (page 180) socket can be controlled by a \texttt{TimeOfDay} (page 190) object so that it is on between the hours of 8am and 8pm:

Using the \texttt{DiskUsage} (page 195) class with \texttt{LEDBarGraph} (page 160) can show your Pi’s disk usage percentage on a bar graph:
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 203) 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 199) of a composite device made up of the nested values of its devices. For example, the value of a `Robot` (page 176) 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

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()` rather than the tool `zip_values()` (page 208) as the `scaled()` (page 206) tool yields values which do not need converting, just zipping. Also note that this use of `zip()` will not work in Python 2, instead use `izip`.

---

53 [https://docs.python.org/3.9/library/functions.html#zip](https://docs.python.org/3.9/library/functions.html#zip)
54 [https://docs.python.org/3.9/library/functions.html#zip](https://docs.python.org/3.9/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. The **pintest** tool is also provided to test the operation of GPIO pins on the board.

### 8.1 pinout

A utility for querying GPIO pin-out information.
gpiozero 2.0 Documentation, Release 2.0

Chapter 8. Command-line Tools

dave@amaterasu:~$ pinout
Description: Raspberry Pi 3B rev 1.2
Revision: a82882
SoC: BCM2837
RAM: 1GB
Storage: MicroSD
USB ports: 4 (of which 0 USB3)
Ethernet ports: 1 (100Mbps max. speed)
Wi-fi: True
Bluetooth: True
Camera ports (CSI): 1
Display ports (DSI): 1

For further information, please refer to https://pinout.xyz/
dave@amaterasu:~$
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 45)) to output information about a remote Pi.

8.1.3 Options

- `-h, --help`
  Show a help message and exit

- `-r REVISION, --revision REVISION`
  Specifies a particular Raspberry Pi board revision code. The default is to autodetect revision of current device by reading `/proc/cpuinfo`

- `-c, --color`
  Force colored output (by default, the output will include ANSI color codes if run in a color-capable terminal). See also `pinout --monochrome` (page 71)

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

- `-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:

```
Description : Raspberry Pi 3B rev 1.2
Revision : a02082
SoC : BCM2837
RAM : 1GB
Storage : MicroSD
USB ports : 4 (of which 0 USB3)
Ethernet ports : 1 (100Mbps max. speed)
Wi-fi : True
Bluetooth : True
Camera ports (CSI) : 1
Display ports (DSI): 1
```

(continues on next page)

56 https://pinout.xyz/
By default, if stdout is a console that supports color, ANSI codes will be used to produce color output. Output can be forced to be `--monochrome` (page 71):

```
$ pinout --monochrome
```

Or forced to be `--color` (page 71), in case you are redirecting to something capable of supporting ANSI codes:

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

To manually specify the revision of Pi you want to query, use `--revision` (page 71). The tool understands both old-style revision codes\(^\text{57}\) (such as for the model B):

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

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

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

---


You can also use the tool with Configuring Remote GPIO (page 45) 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. pinout
8.2 pintest

A utility for testing the GPIO pins on a Raspberry Pi, inspired by pigpio’s gpiotest example script, and wiringPi’s pintest utility.

New in version 2.0: The pintest utility.
8.2. pintest

This program checks the board’s user-accessible GPIO pins. The board’s model is: Raspberry Pi 3B rev 1.2. The following pins are selected for testing:

GPIO12, GPIO10, GPIO12, GPIO18, GPIO26, GPIO20, GPIO25, GPIO19, GPIO18, GPIO09, GPIO23, GPIO11, GPIO6, GPIO21, GPIO15, GPIO7, GPIO4, GPIO5, GPIO13, GPIO14, GPIO16, GPIO1, GPIO17, GPIO27, GPIO24, GPIO22, GPIO22, GPIO08, GPIO3

Please ensure that nothing is connected to any of the pins listed above for the test duration.

Proceed with test? [y/N] y
Testing GPIO12...ok
Testing GPIO10...ok
Testing GPIO2...ok
Testing GPIO18...ok
Testing GPIO26...ok
Testing GPIO20...ok
Testing GPIO25...ok
Testing GPIO19...ok
Testing GPIO10...ok
Testing GPIO9...ok
Testing GPIO23...ok
Testing GPIO11...ok
Testing GPIO6...ok
Testing GPIO21...ok
Testing GPIO15...ok
Testing GPIO7...ok
Testing GPIO4...ok
Testing GPIO5...ok
Testing GPIO13...ok
Testing GPIO14...ok
Testing GPIO16...ok
Testing GPIO1...ok
Testing GPIO17...ok
Testing GPIO27...ok
Testing GPIO24...ok
Testing GPIO22...ok
Testing GPIO8...ok
Testing GPIO3...ok

dave@amaterasu:~$
8.2.1 Synopsis

```
```

8.2.2 Description

A utility for testing the function of GPIOs on a Raspberry Pi. It is possible to damage the GPIOs on a Pi by passing too much current (or voltage in the case of inputs) through them. The `pintest` utility can be used to determine if any of the GPIOs on a Pi are broken.

The utility will test all physically exposed GPIOs (those on the main GPIO header) by default, but you may wish to only test a subset, or to exclude certain GPIOs which can be accomplished with the `pintest --pins` (page 76) or `pintest --skip` (page 76) options.

**Note:** You must ensure that nothing is connected to the GPIOs that you intend to test. By default, the utility will prompt you before proceeding, repeating this warning.

In the event that any GPIO is found to be faulty, it will be reported in the output and the utility will exit with a return code of 1. If all specified GPIOs test fine, the return code is zero.

8.2.3 Options

- **-h, --help**
  show this help message and exit

- **--version**
  Show the program’s version number and exit

- **-p PINS, --pins PINS**
  The pin(s) to test. Can be specified as a comma-separated list of pins. Pin numbers can be given in any form accepted by gpiozero, e.g. 14, GPIO14, BOARD8. The default is to test all pins

- **-s SKIP, --skip SKIP**
  The pin(s) to skip testing. Can be specified as comma-separated list of pins. Pin numbers can be given in any form accepted by gpiozero, e.g. 14, GPIO14, BOARD8. The default is to skip no pins

- **-y, --yes**
  Proceed without prompting

- **-r REVISION, --revision REVISION**
  Force board revision. Default is to autodetect revision of current device. You should avoid this option unless you are very sure the detection is incorrect

8.2.4 Examples

Test all physically exposed GPIOs on the board:

```
$ pintest
```

Test just the I2C GPIOs without prompting:

```
$ pintest --pins 2,3 --yes
```

Exclude the SPI GPIOs from testing:
Note that pin numbers can be given in any form accepted by GPIO Zero, e.g. 14, GPIO14, or BOARD8.

### 8.3 Environment Variables

All utilities provided by GPIO Zero accept the following environment variables:

**GPIOZERO_PIN_FACTORY**

The library to use when communicating with the GPIO pins. Defaults to attempting to load lgpi0, then RPi.GPIO, then piggio, and finally uses a native Python implementation. Valid values include “lgpio”, “rpigpio”, “piggio”, “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 piggio library should attempt to connect to (if the piggio pin factory is being used). Defaults to localhost.

**PIGPIO_PORT**

The port number the piggio library should attempt to connect to (if the piggio 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 125) 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()` to keep the script alive:

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

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

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

def hello():
    print("Hello")

button = Button(2)
button.when_pressed = hello

pause()
```

59 https://docs.python.org/3.9/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 106) is an attribute which reveals whether the button is currently pressed by returning True or False:

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

- **wait_for_press()** (page 106) is a method which blocks the code from continuing until the button is pressed. Also see **wait_for_release()** (page 106):

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

- **when_pressed** (page 107) 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 107):

```
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 105) and *internal devices* (page 189) 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 61) page.

9.3 My event handler isn’t being called

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

```
from gpiozero import Button

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

b = Button(17)
b.when_pressed = pushed()
```
In the case above, when assigning to `when_pressed` (page 107), the thing that is assigned is the `result of calling` the `pushed` function. Because `pushed` doesn’t explicitly return anything, the result is `None`. Hence this is equivalent to doing:

```
b.when_pressed = None
```

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

```
b.when_pressed = pushed
```

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

**Note:** Note that as of v1.5, setting a callback to `None` when it was previously `None` will raise a `Callback-SetToNone` (page 244) warning, with the intention of alerting users when callbacks are set to `None` accidentally. However, if this is intentional, the warning can be suppressed. See the `warnings` 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 221)). 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 77) environment variable. For example:

   ```
   $ GPIOZERO_PIN_FACTORY=pigpio python3
   ```

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

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

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

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

---

60 https://docs.python.org/3.9/library/exceptions.html#None  
61 https://docs.python.org/3.9/library/exceptions.html#None  
62 https://docs.python.org/3.9/library/exceptions.html#None  
63 https://docs.python.org/3.9/library/exceptions.html#None  
64 https://docs.python.org/3.9/library/warnings.html#module-warnings  
65 https://docs.python.org/3.9/library/warnings.html#module-warnings  
66 https://docs.python.org/3.9/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 pinout (page 69) 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 pinout (page 69) 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:

```bash
$ 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 pinout (page 69) 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 to see if it’s been reported before, or check the commits on GitHub since the last release to see if it’s been added. The model determination can be found in gpiozero/pins/data.py.

---

68 https://github.com/gpiozero/gpiozero/issues
69 https://github.com/gpiozero/gpiozero/commits/master
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 199) 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 199) 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 105) class provides a when_held (page 106) 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 106) property). If you want to set when_held (page 106) as well as when_pressed (page 107), you’ll notice that both callbacks will fire. Sometimes, this is acceptable, but often you’ll want to only fire the when_pressed (page 107) 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 107), and instead use when_released (page 107) to work out whether it had been held or just pressed:

```python
from gpiozero import Button

Button.was_held = False

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

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

```
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:

```
>>> from gpiozero import Button
>>> Button.label = ''
>>> btn = Button(2)
>>> btn.label = 'alarm'
>>> def press(btn):
    (continues on next page)
```
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 Zero\(^{70}\), and has been followed by NetworkZero\(^{71}\), guizero\(^{72}\) 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/
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, and all deprecated functionality which will still work in release 2.0 but is scheduled for removal in a future 2.x release.

10.1 Finding and fixing deprecated usage

As of release 2.0, all deprecated functionality will raise `DeprecationWarning` when used. By default, the Python interpreter suppresses these warnings (as they’re only of interest to developers, not users) but you can easily configure different behaviour.

The following example script uses a number of deprecated functions:

```python
import gpiozero
board = gpiozero.pi_info()
for header in board.headers.values():
    for pin in header.pins.values():
        if pin.pull_up:
            print(pin.function, 'is pulled up')
```

Despite using deprecated functionality the script runs happily (and silently) with gpiozero 2.0. To discover what deprecated functions are being used, we add a couple of lines to tell the warnings module that we want “default” handling of `DeprecationWarning`; “default” handling means that the first time an attempt is made to raise this warning at a particular location, the warning’s details will be printed to the console. All future invocations from the same location will be ignored. This saves flooding the console with warning details from tight loops. With this change, the script looks like this:

```python
import warnings
warnings.filterwarnings('default', category=DeprecationWarning)
import gpiozero
board = gpiozero.pi_info()
for header in board.headers.values():
    for pin in header.pins.values():
        if pin.pull_up:
            print(pin.function, 'is pulled up')
```

And produces the following output on the console when run:

---

73 https://docs.python.org/3.9/library/exceptions.html#DeprecationWarning
74 https://docs.python.org/3.9/library/exceptions.html#DeprecationWarning
This tells us which pieces of deprecated functionality are being used in our script, but it doesn’t tell us where in the script they were used. For this, it is more useful to have warnings converted into full blown exceptions. With this change, each time a DeprecationWarning would have been printed, it will instead cause the script to terminate with an unhandled exception and a full stack trace:

```
import gpiozero
import warnings
warnings.filterwarnings('error', category=DeprecationWarning)
board = gpiozero.pi_info()
for header in board.headers.values():
    for pin in header.pins.values():
        if pin.pull_up:
            print(pin.function, 'is pulled up')
```

Now when we run the script it produces the following:

```
Traceback (most recent call last):
 File "/home/dave/projects/home/gpiozero/gpio-zero/foo.py", line 9, in <module>
   if pin.pull_up:
 File "/home/dave/projects/home/gpiozero/gpio-zero/gpiozero/pins/__init__.py", line 899, in pull_up
   warnings.warn(
 DeprecationWarning: PinInfo.pull_up is deprecated; please use PinInfo.pull
```

This tells us that line 10 of our script is using deprecated functionality, and provides a hint of how to fix it. We change line 9 to use the “pull” attribute instead. Now we run again, and this time get the following:

```
Traceback (most recent call last):
 File "/home/dave/projects/home/gpiozero/gpio-zero/foo.py", line 10, in <module>
    print(pin.function, 'is pulled up')
 File "/home/dave/projects/home/gpiozero/gpio-zero/gpiozero/pins/__init__.py", line 889, in function
    warnings.warn(
 DeprecationWarning: PinInfo.function is deprecated; please use PinInfo.name
```

Now we can tell line 10 has a problem, and once again the exception tells us how to fix it. We continue in this fashion until the script looks like this:

```
import gpiozero
import warnings
warnings.filterwarnings('error', category=DeprecationWarning)
board = gpiozero.pi_info()
for header in board.headers.values():
    for pin in header.pins.values():
        if pin.pull == 'up':
            print(pin.name, 'is pulled up')
```

---

[75] https://docs.python.org/3.9/library/exceptions.html#DeprecationWarning
The script now runs to completion, so we can be confident it’s no longer using any deprecated functionality and will run happily even when this functionality is removed in a future 2.x release. At this point, you may wish to remove the `filterwarnings` line as well (or at least comment it out).

### 10.2 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 in the Python documentation.[76]

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, and to ensure that GPIO Zero can run on the version of Python in Debian oldstable at the time of its release.

### 10.3 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).

### 10.4 Deprecated pin-factory aliases removed

Several deprecated aliases for pin factories, which could be specified by the `GPIOZERO_PIN_FACTORY` (page 77) 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”

In other words, you can no longer use the following when invoking your script:

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

Instead, you should use the following:

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

### 10.5 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 170) 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 2.0, 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:

[76] https://docs.python.org/3/howto/pyporting.html
[77] https://docs.python.org/3/
from gpiozero import PiLiter
l = PiLiter(pwm=True)

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

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

## 10.6 Robots take Motors, and PhaseEnableRobot is deprecated

The GPIO Zero 1.x API specified that a `Robot` (page 176) was constructed with two tuples that were in turn used to construct two `Motor` (page 134) instances. The 2.x API replaces this with simply passing in the `Motor` (page 134), or `PhaseEnableMotor()` (page 136) 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:

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

This should be changed to the following:

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

Likewise, if you are currently using `PhaseEnableRobot()` (page 178) your code may look like this:

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

This should be changed to the following:

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

This change came about because the `Motor` (page 134) 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 176) which was inconsistent. Furthermore, `PhaseEnableRobot()` (page 178) was more or less a redundant duplicate of `Robot` (page 176) but was lacking a couple of features added to `Robot` (page 176) 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 176) 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 176) constructor) will remain as deprecated functionality for gpiozero 2.0, but will be removed in a future 2.x release. `PhaseEnableRobot()` (page 178) remains as a stub function which simply returns a `Robot` (page 176) instance, but this will be removed in a future 2.x release.
10.7 PiBoardInfo, HeaderInfo, PinInfo

The `PiBoardInfo` (page 217) class, and the associated `HeaderInfo` (page 218) and `PinInfo` (page 219) classes have undergone a major re-structuring. This is partly because some of the prior terminology was confusing (e.g. the meaning of `PinInfo.function` (page 219) and `Pin.function` (page 229) clashed), and partly because with the addition of the “Igpio” factory it’s entirely possible to use gpiozero on non-Pi boards (although at present the `pins.Igpio.LGPIOFactory` (page 235) is still written assuming it is only ever used on a Pi).

As a result the following classes, methods, and attributes are deprecated (not yet removed, but will be in a future release within the 2.x series):

- `Factory.pi_info` is deprecated in favour of `Factory.board_info` (page 227) which returns a `BoardInfo` instead of `PiBoardInfo` (page 217) (which is now a subclass of the former).
- `PinInfo.pull_up` (page 219) is deprecated in favour of `PinInfo.pull` (page 219).
- `PinInfo.function` (page 219) is deprecated in favour of `PinInfo.name` (page 219).
- `BoardInfo.physical_pins()`, `BoardInfo.physical_pin()`, and `BoardInfo.pulled_up()`, are all deprecated in favour of a combination of the new `BoardInfo.find_pin()` and the attributes mentioned above.
- `PiPin.number` is deprecated in favour of `Pin.info.name`. 
CHAPTER
ELEVEN

MIGRATING FROM RPI.GPIO

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

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

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

As well as specific device classes, we provide base classes InputDevice (page 121) and OutputDevice (page 144). 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 223).

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)
```

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() (page 126) method.

---

78 https://pypi.org/project/RPi.GPIO/
79 https://pypi.org/project/RPi.GPIO/
OutputDevice (page 144) 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 (page 125). 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 (page 126).

## 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:

```python
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:

\[\text{https://pypi.org/project/RPi.GPIO/}\]
The equivalent in GPIO Zero:

```python
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:

```python
from gpiozero import Button

btn = Button(4, pull_up=False)

btn.wait_for_press()

print("button was pressed")
```

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

```python
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:

```python
import RPi.GPIO as GPIO

GPIO.setmode(GPIO.BCM)
GPIO.setwarnings(False)

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

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

GPIO.setup(4, GPIO.IN, GPIO.PUD_UP)

GPIO.add_event_detect(4, GPIO.FALLING, pressed)
GPIO.add_event_detect(4, GPIO.RISING, released)
```

In GPIO Zero, you assign the `when_pressed` (page 107) and `when_released` (page 107) properties to set up callbacks on those actions.
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 106) 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 121) 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 105). Other input devices have similar property and method names. There is commonality in naming at base level, such as InputDevice.is_active (page 122), which is aliased in a device class, such as Button.is_pressed (page 106) and LightSensor.light_detected (page 112).

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 128), LEDBoard (page 157), DistanceSensor (page 113), Motor (page 134) and Robot (page 176).

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 113) object, specifying the trigger and echo pins, and you would read the DistanceSensor.distance (page 114) property which automatically calculates the distance within the implementation of the class.

The Motor (page 134) class controls two output pins to drive the motor forwards or backwards. The Robot (page 176) 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 157) class takes an arbitrary number of pins, each controlling a single LED. The resulting LEDBoard (page 157) 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 37).

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:

import RPi.GPIO as GPIO
from time import sleep

(continues on next page)
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 127) has a `blink()` (page 127) method which can be used the same way as LED (page 125)'s `blink()` (page 126) method, but its PWM capabilities allow for `fade_in` and `fade_out` options to be provided. There is also the `pulse()` (page 127) 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 134), `Servo` (page 137) and `AngularServo` (page 139) classes for information on those. Motor (page 134) and Robot (page 176) 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 127) 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 possible by use of the `close()` (page 199) 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 83)
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 217):

```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=...)
```

Read more about what `PiBoardInfo` (page 217) provides.

11.7 More

GPIO Zero provides more than just GPIO device support, it includes some support for SPI devices (page 147) 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 192), TimeOfDay (page 190), PingServer (page 191) and LoadAverage (page 193) 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 223) and Configuring Remote GPIO (page 45) 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 225).

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 61).

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 79)
- *Why do I get PinFactoryFallback warnings when I import gpiozero?* (page 81)
- *What’s the gpiozero equivalent of GPIO.cleanup()?* (page 83)
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[^82] explaining your reasoning clearly.

12.2 Bugs

Please submit bug reports by opening an issue[^83] explaining the problem clearly using code examples.

12.3 Documentation

The documentation source lives in the docs[^84] 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 255). 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.

[^82]: https://github.com/gpiozero/gpiozero/issues/new
[^83]: https://github.com/gpiozero/gpiozero/issues/new
[^84]: https://github.com/gpiozero/gpiozero/tree/master/docs
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\(^\text{85}\), the 1.6.2 release (2021-03-18) is the last to support Python 2.

\(^{85}\text{http://legacy.python.org/dev/peps/pep-0373/}\)
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 125) and follow its heritage backward to `DigitalOutputDevice` (page 141). Follow that back to `OutputDevice` (page 144) 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:

```bash
$ 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:

```bash
$ 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):

```bash
(gpiozero) $ pip install rpi.gpio pigpio
```

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

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

To pull the latest changes from git into your clone and update your installation:
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 101) 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[86] 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).

[86] https://launchpad.net/~deadsnakes/+archive/ubuntu/ppa
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 237), which is also useful for manual testing, for example in the Python shell or another REPL. See the section on `Mock pins` (page 225) in the `API - Pins` (page 221) chapter for more information.
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(*args, **kwargs)

Extends DigitalInputDevice (page 119) and represents a simple push button or switch.

Connect one side of the button to a ground pin, and the other to any GPIO pin. Alternatively, connect one side of the button to the 3V3 pin, and the other to any GPIO pin, then set pull_up to False in the Button (page 105) 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

- **pin** (int or str) – The GPIO pin which the button is connected to. See Pin Numbering (page 3) for valid pin numbers. If this is None a GPIODeviceError (page 241) will be raised.

- **pull_up** (bool or None) – If True (the default), the GPIO pin will be pulled high by default. In this case, connect the other side of the button to ground. If False, the GPIO pin will be pulled low by default. In this case, connect the other side of the button to 3V3. If None, the pin will be floating, so it must be externally pulled up or down and the active_state parameter must be set accordingly.

- **active_state** (bool or None) – See description under InputDevice (page 121) for more information.
• **bounce_time** (*float* or *None*) – If *None* (the default), no software bounce compensation will be performed. Otherwise, this is the length of time (in seconds) that the component will ignore changes in state after an initial change.

• **hold_time** (*float*) – The length of time (in seconds) to wait after the button is pushed, until executing the *when_held* (page 106) handler. Defaults to 1.

• **hold_repeat** (*bool*) – If *True*, the *when_held* (page 106) handler will be repeatedly executed as long as the device remains active, every *hold_time* seconds. If *False* (the default) the *when_held* (page 106) handler will be only be executed once per hold.

• **pin_factory** (*Factory* (page 226) or *None*) – See API - Pins (page 221) for more information (this is an advanced feature which most users can ignore).

*wait_for_press* (*timeout=None*)

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

**Parameters**

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

*wait_for_release* (*timeout=None*)

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

**Parameters**

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

**property 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 106) event rather than when the device activated, in contrast to *active_time* (page 201). If the device is not currently held, this is *None*.

**property hold_repeat**

If *True*, *when_held* (page 106) will be executed repeatedly with *hold_time* (page 106) seconds between each invocation.

**property hold_time**

The length of time (in seconds) to wait after the device is activated, until executing the *when_held* (page 106) handler. If *hold_repeat* (page 106) is True, this is also the length of time between invocations of *when_held* (page 106).

**property is_held**

When *True*, the device has been active for at least *hold_time* (page 106) seconds.

**property is_pressed**

Returns *True* if the device is currently active and *False* otherwise. This property is usually derived from *value* (page 106). Unlike *value* (page 106), this is *always* a boolean.

**property pin**

The *Pin* (page 227) that the device is connected to. This will be *None* if the device has been closed (see the *close()* (page 199) method). When dealing with GPIO pins, query *pin.number* to discover the GPIO pin (in BCM numbering) that the device is connected to.

**property pull_up**

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

**property 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 \textit{hold_time} (page 106) 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 \texttt{None}^{113} (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 \texttt{None}^{114} (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.

Set this property to \texttt{None}^{115} (the default) to disable the event.
14.1.2 LineSensor (TRCT5000)

```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 241) will be raised.

- **pull_up** *(bool or None)* – See description under InputDevice (page 121) for more information.

- **active_state** *(bool or None)* – See description under InputDevice (page 121) 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 121) property, and all appropriate events will be fired.

- **partial** *(bool)* – When False (the default), the object will not return a value for is_active (page 121) until the internal queue has filled with values. Only set this to True if you require values immediately after object construction.

- **pin_factory** *(Factory or None)* – See API - Pins (page 221) 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.
property pin
The Pin (page 227) that the device is connected to. This will be None if the device has been closed (see the close() (page 199) method). When dealing with GPIO pins, query pin.number to discover the GPIO pin (in BCM numbering) that the device is connected to.

property 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.

14.1.3 MotionSensor (D-SUN PIR)

class gpiozero.MotionSensor(*args, **kwargs)
Extends SmoothedInputDevice (page 120) and represents a passive infra-red (PIR) motion sensor like the sort found in the CamJam #2 EduKit.
A typical PIR device has a small circuit board with three pins: VCC, OUT, and GND. VCC should be connected to a 5V pin, GND to one of the ground pins, and finally OUT to the GPIO specified as the value of the pin parameter in the constructor.
The following code will print a line of text when motion is detected:

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

116 http://camjam.me/?page_id=1035
117 https://docs.python.org/3.9/library/functions.html#int
118 https://docs.python.org/3.9/library/stdtypes.html#str
119 https://docs.python.org/3.9/library/constants.html#None
120 https://docs.python.org/3.9/library/functions.html#bool
121 https://docs.python.org/3.9/library/functions.html#bool
122 https://docs.python.org/3.9/library/functions.html#int
123 https://docs.python.org/3.9/library/functions.html#float
124 https://docs.python.org/3.9/library/functions.html#float
125 https://docs.python.org/3.9/library/constants.html#False
126 https://docs.python.org/3.9/library/constants.html#True
127 https://docs.python.org/3.9/library/constants.html#None
128 https://docs.python.org/3.9/library/functions.html#float
129 https://docs.python.org/3.9/library/constants.html#None
130 https://docs.python.org/3.9/library/functions.html#float
131 https://docs.python.org/3.9/library/constants.html#None
132 https://docs.python.org/3.9/library/constants.html#None
133 https://docs.python.org/3.9/library/constants.html#None
134 https://docs.python.org/3.9/library/constants.html#None
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 241) will be raised.

- **pull_up** *(bool or None)* – See description under *InputDevice* (page 121) for more information.

- **active_state** *(bool or None)* – See description under *InputDevice* (page 121) for more information.

- **queue_len** *(int)* – The length of the queue used to store values read from the sensor. This defaults to 1 which effectively disables the queue. If your motion sensor is particularly “twitchy” you may wish to increase this value.

- **sample_rate** *(float)* – The number of values to read from the device (and append to the internal queue) per second. Defaultsto 10.

- **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 121) property, and all appropriate events will be fired.

- **partial** *(bool)* – When *False* (the default), the object will not return a value for *is_active* (page 121) until the internal queue has filled with values. Only set this to *True* if you require values immediately after object construction.

- **pin_factory** *(Factory or None)* – See *API - Pins* (page 221) for more information (this is an advanced feature which most users can ignore).

### wait_for_motion *(timeout=None)*

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

#### Parameters

- **timeout** *(float 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_no_motion *(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.

### property motion_detected

Returns *True* if the *value* (page 121) currently exceeds *threshold* (page 121) and *False* otherwise.

### property pin

The *Pin* (page 227) that the device is connected to. This will be *None* if the device has been closed (see the *close()* (page 199) method). When dealing with GPIO pins, query *pin.number* to discover the GPIO pin (in BCM numbering) that the device is connected to.

### property 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 `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 `None` (the default) to disable the event.

### 14.1.4 LightSensor (LDR)

**class** `gpiozero.LightSensor(*args, **kwargs)`

Extends `SmoothedInputDevice` (page 120) 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` or `str`) – The GPIO pin which the sensor is attached to. See Pin Numbering (page 3) for valid pin numbers. If this is `None` a `GPIODeviceError` (page 241) will be raised.
- **queue_len** (`int`) – The length of the queue used to store values read from the circuit. This defaults to 5.
- **charge_time_limit** (`float`) – If the capacitor in the circuit takes longer than this length of time to charge, it is assumed to be dark. The default (0.01 seconds) is appropriate for a 1µF capacitor coupled with the LDR from the CamJam #2 EduKit. You may need to adjust this value for different valued capacitors or LDRs.
- **threshold** (*float*): Defaults to 0.1. When the average of all values in the internal queue rises above this value, the area will be considered “light”, and all appropriate events will be fired.

- **partial** (*bool*): When *False* (the default), the object will not return a value for *is_active* (page 121) until the internal queue has filled with values. Only set this to *True* if you require values immediately after object construction.

- **pin_factory** (*Factory* or *None*): See API - Pins (page 221) for more information (this is an advanced feature which most users can ignore).

### wait_for_dark *(timeout=None)*

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

**Parameters**

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

### wait_for_light *(timeout=None)*

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

**Parameters**

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

### property light_detected

Returns *True* if the value (page 121) currently exceeds *threshold* and *False* otherwise.

### property pin

The *Pin* (page 227) that the device is connected to. This will be *None* if the device has been closed (see the close() (page 199) method). When dealing with GPIO pins, query *pin.number* to discover the GPIO pin (in BCM numbering) that the device is connected to.

### property 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* (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* (the default) to disable the event.
14.1.5 DistanceSensor (HC-SR04)

class gpiozero.DistanceSensor(*args, **kwargs)

Extends SmoothedInputDevice (page 120) and represents an HC-SR04 ultrasonic distance sensor, as found in the CamJam #3 EduKit. The distance sensor requires two GPIO pins: one for the trigger (marked TRIG on the sensor) and another for the echo (marked ECHO on the sensor). However, a voltage divider is required to ensure the 5V from the ECHO pin doesn’t damage the Pi. Wire your sensor according to the following instructions:

1. Connect the GND pin of the sensor to a ground pin on the Pi.
2. Connect the TRIG pin of the sensor a GPIO pin.
3. Connect one end of a 330Ω resistor to the ECHO pin of the sensor.
4. Connect one end of a 470Ω resistor to the GND pin of the sensor.
5. Connect the free ends of both resistors to another GPIO pin. This forms the required voltage divider.
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 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
Zer. See Changing the pin factory (page 223) for further information.

Parameters

- **echo** (int or str) – The GPIO pin which the ECHO pin is connected to. See Pin Numbering (page 3) for valid pin numbers. If this is None a GPIODeviceError (page 241) will be raised.

- **trigger** (int or str) – The GPIO pin which the TRIG pin is connected to. See Pin Numbering (page 3) for valid pin numbers. If this is None a GPIODeviceError (page 241) will be raised.

- **queue_len** (int) – The length of the queue used to store values read from the sensor. This defaults to 9.

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

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

- **partial** (bool) – When False (the default), the object will not return a value for is_active (page 121) until the internal queue has filled with values. Only set this to True if you require values immediately after object construction.

- **pin_factory** (Factory or None) – See API - Pins (page 221) 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.

**property 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 114).

**property echo**

Returns the Pin (page 227) that the sensor’s echo is connected to. This is simply an alias for the usual pin (page 123) attribute.

**property 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 121) attribute. When distance (page 114) is equal to max_distance (page 114), value (page 121) will be 1.

**property threshold_distance**

The distance, measured in meters, that will trigger the when_in_range (page 115) and when_out_of_range (page 115) events when crossed. This is simply a meter-scaled variant of the usual threshold (page 121) attribute.
property trigger
Returns the Pin (page 227) that the sensor’s trigger is connected to.

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

when_in_range
The function to run when the device changes state from active to inactive.
This can be set to a function which accepts no (mandatory) parameters, or a Python function which accepts a single mandatory parameter (with as many optional parameters as you like). If the function accepts a single mandatory parameter, the device that deactivated 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

class gpiozero.RotaryEncoder(*args, **kwargs)
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:

175 http://camjam.me/?page_id=1035
176 https://en.wikipedia.org/wiki/Voltage_divider
177 https://en.wikipedia.org/wiki/Voltage_divider
178 https://docs.python.org/3.9/library/functions.html#int
179 https://docs.python.org/3.9/library/stdtypes.html#str
180 https://docs.python.org/3.9/library/constants.html#None
181 https://docs.python.org/3.9/library/functions.html#int
182 https://docs.python.org/3.9/library/stdtypes.html#str
183 https://docs.python.org/3.9/library/constants.html#None
184 https://docs.python.org/3.9/library/functions.html#int
185 https://docs.python.org/3.9/library/functions.html#float
186 https://docs.python.org/3.9/library/functions.html#float
187 https://docs.python.org/3.9/library/functions.html#bool
188 https://docs.python.org/3.9/library/constants.html#False
189 https://docs.python.org/3.9/library/constants.html#True
190 https://docs.python.org/3.9/library/functions.html#float
191 https://docs.python.org/3.9/library/constants.html#None
192 https://docs.python.org/3.9/library/functions.html#float
193 https://docs.python.org/3.9/library/constants.html#None
194 https://docs.python.org/3.9/library/constants.html#None
195 https://docs.python.org/3.9/library/constants.html#None
>>> 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 117) from 0 to 1, or counter-clockwise from 0 to -1. If this is 0, then the encoder’s value (page 117) never changes, but you can still read steps (page 116) 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 116) 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 116) reaches positive or negative max_steps it wraps around by negation. Defaults to False.
- **pin_factory** *(Factory or None)* – See API - Pins (page 221) for more information (this is an advanced feature which most users can ignore).

**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.

**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.

**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.

**property max_steps**

The number of discrete steps the rotary encoder takes to move value (page 117) 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.
property 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 116). It will not advance beyond positive or negative `max_steps` (page 116), unless `wrap` (page 117) is `True` in which case it will roll around by negation. If `max_steps` (page 116) is zero then steps are not limited at all, and will increase infinitely in either direction, but `value` (page 117) 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 116) attribute (and `value` (page 117) by corollary) is writable.

property threshold_steps

The minimum and maximum number of steps between which `is_active` will return `True`. Defaults to `(0, 0)`.

property 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 116) into the range from negative `max_steps` (page 116) to positive `max_steps` (page 116).

Note that, in contrast to most other input devices, because the rotary encoder has no absolute position the `value` (page 117) 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` (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` (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` (the default) to disable the event.

property wrap

If `True`, when `value` (page 117) reaches its limit (-1 or 1), it “wraps around” to the opposite limit. When `False`, the value (and the corresponding `steps` (page 116) attribute) simply don’t advance beyond their limits.
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):

![Base Class Hierarchy Diagram]

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

---

197 https://docs.python.org/3.9/library/functions.html#int
198 https://docs.python.org/3.9/library/stdtypes.html#str
199 https://docs.python.org/3.9/library/functions.html#int
200 https://docs.python.org/3.9/library/stdtypes.html#str
201 https://docs.python.org/3.9/library/functions.html#float
202 https://docs.python.org/3.9/library/constants.html#None
203 https://docs.python.org/3.9/library/functions.html#int
204 https://docs.python.org/3.9/library/stdtypes.html#int
205 https://docs.python.org/3.9/library/functions.html#int
206 https://docs.python.org/3.9/library/constants.html#True
207 https://docs.python.org/3.9/library/functions.html#bool
208 https://docs.python.org/3.9/library/constants.html#False
209 https://docs.python.org/3.9/library/constants.html#int
210 https://docs.python.org/3.9/library/functions.html#float
211 https://docs.python.org/3.9/library/constants.html#None
212 https://docs.python.org/3.9/library/functions.html#float
213 https://docs.python.org/3.9/library/constants.html#None
214 https://docs.python.org/3.9/library/functions.html#float
215 https://docs.python.org/3.9/library/constants.html#None
216 https://docs.python.org/3.9/library/constants.html#True
217 https://docs.python.org/3.9/library/constants.html#True
218 https://docs.python.org/3.9/library/constants.html#None
219 https://docs.python.org/3.9/library/constants.html#None
220 https://docs.python.org/3.9/library/constants.html#False
221 https://docs.python.org/3.9/library/constants.html#True
222 https://docs.python.org/3.9/library/constants.html#False
14.2.1 DigitalInputDevice

class gpiozero.DigitalInputDevice(*args, **kwargs)

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

This class extends InputDevice (page 121) 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 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 241) will be raised.
- **pull_up** (bool or None) – See description under InputDevice (page 121) for more information.
- **active_state** (bool or None) – See description under InputDevice (page 121) for more information.
- **bounce_time** (float or None) – Specifies the length of time (in seconds) that the component will ignore changes in state after an initial change. This defaults to None which indicates that no bounce compensation will be performed.
- **pin_factory** (Factory or None) – See API - Pins (page 221) 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.

property active_time

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

property inactive_time

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

property 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.

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

14.2.2 SmoothedInputDevice
class gpiozero.SmoothedInputDevice(*args, **kwargs)
Represents a generic input device which takes its value from the average of a queue of historical values.

This class extends InputDevice (page 121) 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 121) 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 241) will be raised.
- **pull_up**(bool or None) – See description under InputDevice (page 121) for more information.
- **active_state**(bool or None) – See description under InputDevice (page 121) 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 121) property) will block until the queue has filled. If...
True\textsuperscript{248}, 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 \texttt{statistics.median()}\textsuperscript{249} 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** (\texttt{frozenset}\textsuperscript{250} or \texttt{None}) – The set of values which the queue should ignore, if returned from querying the device’s value.

- **pin\_factory** (\texttt{Factory} (page 226) or \texttt{None}) – See API - Pins (page 221) for more information (this is an advanced feature which most users can ignore).

property **is\_active**

Returns \texttt{True}\textsuperscript{251} if the \texttt{value} (page 121) currently exceeds \texttt{threshold} (page 121) and \texttt{False}\textsuperscript{252} otherwise.

property **partial**

If \texttt{False}\textsuperscript{253} (the default), attempts to read the \texttt{value} (page 121) or \texttt{is\_active} (page 121) properties will block until the queue has filled.

property **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.

property **threshold**

If \texttt{value} (page 121) exceeds this amount, then \texttt{is\_active} (page 121) will return \texttt{True}\textsuperscript{254}.

property **value**

Returns the average of the values in the internal queue. This is compared to \texttt{threshold} (page 121) to determine whether \texttt{is\_active} (page 121) is \texttt{True}\textsuperscript{255}.

### 14.2.3 InputDevice

**class** `gpiozero.InputDevice` (*\*\*\*args, **\*\*kwargs*)

Represents a generic GPIO input device.

This class extends \texttt{GPIODevice} (page 122) to add facilities common to GPIO input devices. The constructor adds the optional \texttt{pull\_up} parameter to specify how the pin should be pulled by the internal resistors. The \texttt{is\_active} (page 122) property is adjusted accordingly so that \texttt{True}\textsuperscript{256} still means active regardless of the \texttt{pull\_up} setting.

**Parameters**

\texttt{https://docs.python.org/3.9/library/functions.html#int}
\texttt{https://docs.python.org/3.9/library/stdtypes.html#str}
\texttt{https://docs.python.org/3.9/library/constants.html#None}
\texttt{https://docs.python.org/3.9/library/functions.html#bool}
\texttt{https://docs.python.org/3.9/library/functions.html#float}
\texttt{https://docs.python.org/3.9/library/functions.html#int}
\texttt{https://docs.python.org/3.9/library/functions.html#float}
\texttt{https://docs.python.org/3.9/library/functions.html#bool}
\texttt{https://docs.python.org/3.9/library/constants.html#False}
\texttt{https://docs.python.org/3.9/library/constants.html#True}
\texttt{https://docs.python.org/3.9/library/statistics.html#statistics.median}
\texttt{https://docs.python.org/3.9/library/stdtypes.html#frozenset}
\texttt{https://docs.python.org/3.9/library/constants.html#True}
\texttt{https://docs.python.org/3.9/library/constants.html#False}
\texttt{https://docs.python.org/3.9/library/constants.html#False}
\texttt{https://docs.python.org/3.9/library/constants.html#True}
\texttt{https://docs.python.org/3.9/library/constants.html#True}
• **pin** *(int\(^{257}\) or str\(^{258}\)) – The GPIO pin that the device is connected to. See *Pin Numbering* (page 3) for valid pin numbers. If this is `None\(^{259}\)` a *GPIODeviceError* (page 241) will be raised.

• **pull_up** *(bool\(^{260}\) or None)* – If `True\(^{261}\)`, the pin will be pulled high with an internal resistor. If `False\(^{262}\)` (the default), the pin will be pulled low. If `None\(^{263}\)`, 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\(^{264}\) or None)* – If `True\(^{265}\)`, when the hardware pin state is `HIGH`, the software pin is `HIGH`. If `False\(^{266}\)` 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\(^{267}\)`). When `pull_up` is `True\(^{268}\)` or `False\(^{269}\)`, the active state is automatically set to the proper value.

• **pin_factory** *(Factory (page 226) or None)* – See *API - Pins* (page 221) for more information (this is an advanced feature which most users can ignore).

**property is_active**

Returns `True\(^{270}\)` if the device is currently active and `False\(^{271}\)` otherwise. This property is usually derived from `value` (page 122). Unlike `value` (page 122), this is always a boolean.

**property pull_up**

If `True\(^{272}\)`, the device uses a pull-up resistor to set the GPIO pin “high” by default.

**property 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 (*args, **kwargs*)

Extends *Device* (page 199). 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 123)).

**Parameters**

• **pin** *(int\(^{273}\) or str\(^{274}\)) – The GPIO pin that the device is connected to. See *Pin Numbering* (page 3) for valid pin numbers. If this is `None\(^{275}\)` a *GPIODeviceError* (page 241) will be raised. If the pin is already in use by another device, *GPIOPinInUse* (page 241) 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 199) 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()
...    ...
```

**property closed**
Returns `True` if the device is closed (see the `close()` (page 122) method). Once a device is closed you can no longer use any other methods or properties to control or query the device.

**property pin**
The `Pin` (page 227) that the device is connected to. This will be `None` if the device has been closed (see the `close()` (page 199) method). When dealing with GPIO pins, query `pin.number` to discover the GPIO pin (in BCM numbering) that the device is connected to.

**property 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.
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

**class** `gpiozero.LED(*args, **kwargs)`

Extends `DigitalOutputDevice` (page 141) and represents a light emitting diode (LED).

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

The following example will light the LED:

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

**Parameters**

- **pin** *(int 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 241) 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 221) 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.

**property is\_lit**

Returns **True** if the device is currently active and **False** otherwise. This property is usually derived from **value** (page 126). Unlike **value** (page 126), this is always a boolean.

**property pin**

The **Pin** (page 227) that the device is connected to. This will be **None** if the device has been closed (see the **close()** (page 199) method). When dealing with GPIO pins, query **pin.number** to discover the GPIO pin (in BCM numbering) that the device is connected to.

**property value**

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

---

279 https://docs.python.org/3.9/library/functions.html#int
280 https://docs.python.org/3.9/library/stdtypes.html#str
281 https://docs.python.org/3.9/library/constants.html#None
282 https://docs.python.org/3.9/library/functions.html#bool
283 https://docs.python.org/3.9/library/constants.html#True
284 https://docs.python.org/3.9/library/constants.html#False
285 https://docs.python.org/3.9/library/functions.html#bool
286 https://docs.python.org/3.9/library/constants.html#False
287 https://docs.python.org/3.9/library/constants.html#None
288 https://docs.python.org/3.9/library/constants.html#True
289 https://docs.python.org/3.9/library/constants.html#None
290 https://docs.python.org/3.9/library/constants.html#False
291 https://docs.python.org/3.9/library/functions.html#float
292 https://docs.python.org/3.9/library/functions.html#float
293 https://docs.python.org/3.9/library/functions.html#int
294 https://docs.python.org/3.9/library/constants.html#None
295 https://docs.python.org/3.9/library/constants.html#False
296 https://docs.python.org/3.9/library/constants.html#False
297 https://docs.python.org/3.9/library/constants.html#True
298 https://docs.python.org/3.9/library/constants.html#None
299 https://docs.python.org/3.9/library/constants.html#None
15.1.2 PWMLED

```python
class gpiozero.PWMLED(*args, **kwargs)
Extends PWMOutputDevice (page 142) 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 241) will be raised.
- **active_high** (bool) – If True (the default), the on() (page 127) method will set the GPIO to HIGH. If False, the on() (page 127) method will set the GPIO to LOW (the off() (page 127) 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 221) 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).

```python
toggle()
```

Toggle the state of the device. If the device is currently off (value (page 128) is 0.0), this changes it to “fully” on (value (page 128) is 1.0). If the device has a duty cycle (value (page 128)) of 0.1, this will toggle it to 0.9, and so on.

```python
property is_lit
```

Returns True if the device is currently active (value (page 128) is non-zero) and False otherwise.

```python
property pin
```

The Pin (page 227) that the device is connected to. This will be None if the device has been closed (see the close() (page 199) method). When dealing with GPIO pins, query pin.number to discover the GPIO pin (in BCM numbering) that the device is connected to.

```python
property 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

**Class** `gpiozero.RGBLED(*args, **kwargs)`

Extends Device (page 199) 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
https://docs.python.org/3.9/library/functions.html#int
https://docs.python.org/3.9/library/stdtypes.html#str
https://docs.python.org/3.9/library/converters.html#None
https://docs.python.org/3.9/library/converters.html#bool
https://docs.python.org/3.9/library/converters.html#True
https://docs.python.org/3.9/library/converters.html#False
https://docs.python.org/3.9/library/converters.html#float
https://docs.python.org/3.9/library/converters.html#int
https://docs.python.org/3.9/library/converters.html#float
https://docs.python.org/3.9/library/converters.html#float
https://docs.python.org/3.9/library/converters.html#float
https://docs.python.org/3.9/library/converters.html#int
https://docs.python.org/3.9/library/converters.html#None
https://docs.python.org/3.9/library/converters.html#bool
https://docs.python.org/3.9/library/converters.html#True
https://docs.python.org/3.9/library/converters.html#False
https://docs.python.org/3.9/library/converters.html#int
https://docs.python.org/3.9/library/converters.html#float
https://docs.python.org/3.9/library/converters.html#float
https://docs.python.org/3.9/library/converters.html#float
https://docs.python.org/3.9/library/converters.html#float
https://docs.python.org/3.9/library/converters.html#int
https://docs.python.org/3.9/library/converters.html#None
https://docs.python.org/3.9/library/converters.html#bool
https://docs.python.org/3.9/library/converters.html#True
https://docs.python.org/3.9/library/converters.html#False
https://docs.python.org/3.9/library/converters.html#int
from gpiozero import RGBLED

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

The colorzero library is also supported:

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 241) 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 127) instances for each component of the RGBLED. If *False*, construct regular *LED* (page 125) instances, which prevents smooth color graduations.
- **pin_factory** (*Factory* or *None*) – See API - Pins (page 221) 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).

**toggle()**

Toggle the state of the device. If the device is currently off \((value\) (page 130) is \((0, 0, 0)\)), this changes it to “fully” on \((value\) (page 130) is \((1, 1, 1)\)). If the device has a specific color, this method inverts the color.

**property blue**

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

**property color**

Represents the color of the LED as a **Color** object.

**property green**

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

**property is_lit**

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

**property red**

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

**property 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

class gpiozero.Buzzer(*args, **kwargs)

Extends DigitalOutputDevice (page 141) and represents a digital buzzer component.

**Note:** This interface is only capable of simple on/off commands, and is not capable of playing a variety of tones (see TonalBuzzer (page 133)).

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

The following example will sound the buzzer:

```python
from gpiozero import Buzzer
```

(continues on next page)
bz = Buzzer(3)
bz.on()

Parameters

- **pin** (int or str) – The GPIO pin which the buzzer is connected to. See Pin Numbering (page 3) for valid pin numbers. If this is None a GPIODeviceError (page 241) will be raised.

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

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

- **pin_factory** (Factory or None) – See API - Pins (page 221) for more information (this is an advanced feature which most users can ignore).

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

Parameters

- **on_time** (float) – 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.

**property is_active**
Returns True if the device is currently active and False otherwise. This property is usually derived from value (page 132). Unlike value (page 132), this is always a boolean.

**property pin**
The Pin (page 227) that the device is connected to. This will be None if the device has been closed (see the close() (page 199) method). When dealing with GPIO pins, query pin.number to discover the GPIO pin (in BCM numbering) that the device is connected to.

**property 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(*args, **kwargs)

Extends CompositeDevice (page 187) 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 241) 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 or None) – See API - Pins (page 221) 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 236).

play (tone)

Play the given tone. This can either be an instance of Tone (page 215) or can be anything that could be used to construct an instance of Tone (page 215).

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 134) to None.

378 https://docs.python.org/3.9/library/functions.html#int
379 https://docs.python.org/3.9/library/stdtypes.html#str
380 https://docs.python.org/3.9/library/constants.html#None
381 https://docs.python.org/3.9/library/functions.html#bool
382 https://docs.python.org/3.9/library/constants.html#True
383 https://docs.python.org/3.9/library/constants.html#False
384 https://docs.python.org/3.9/library/functions.html#bool
385 https://docs.python.org/3.9/library/constants.html#False
386 https://docs.python.org/3.9/library/constants.html#True
387 https://docs.python.org/3.9/library/constants.html#None
388 https://docs.python.org/3.9/library/functions.html#float
389 https://docs.python.org/3.9/library/functions.html#float
390 https://docs.python.org/3.9/library/functions.html#int
391 https://docs.python.org/3.9/library/constants.html#None
392 https://docs.python.org/3.9/library/functions.html#bool
393 https://docs.python.org/3.9/library/constants.html#True
394 https://docs.python.org/3.9/library/constants.html#False
395 https://docs.python.org/3.9/library/constants.html#True
396 https://docs.python.org/3.9/library/constants.html#False
397 https://docs.python.org/3.9/library/constants.html#None
property is_active
Returns True if the buzzer is currently playing, otherwise False.

property max_tone
The highest tone that the buzzer can play, i.e. the tone played when value (page 134) is 1.

property mid_tone
The middle tone available, i.e. the tone played when value (page 134) is 0.

property min_tone
The lowest tone that the buzzer can play, i.e. the tone played when value (page 134) is -1.

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

property tone
Returns the Tone (page 215) that the buzzer is currently playing, or None if the buzzer is silent. This property can also be set to play the specified tone.

property 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.

15.1.6 Motor

class gpiozero.Motor(*args, **kwargs)
Extends CompositeDevice (page 187) and represents a generic motor connected to a bi-directional motor driver circuit (i.e. an H-bridge).

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

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

```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 241) will be raised.

---

398 https://docs.python.org/3.9/library/functions.html#int
399 https://docs.python.org/3.9/library/stdtypes.html#str
400 https://docs.python.org/3.9/library/constants.html#None
401 https://docs.python.org/3.9/library/functions.html#float
402 https://docs.python.org/3.9/library/constants.html#None
403 https://docs.python.org/3.9/library/functions.html#int
404 https://docs.python.org/3.9/library/stdtypes.html#str
405 https://docs.python.org/3.9/library/functions.html#int
406 https://docs.python.org/3.9/library/constants.html#None
407 https://docs.python.org/3.9/library/constants.html#None
408 https://docs.python.org/3.9/library/constants.html#True
409 https://docs.python.org/3.9/library/constants.html#None
410 https://docs.python.org/3.9/library/constants.html#None
• **backward** *(int\(^{416}\) or str\(^{417}\))* – 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\(^{418}\) a **GPIODeviceError** (page 241) will be raised.

• **enable** *(int\(^{419}\) or str\(^{420}\) 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\(^{421}\))* – If True\(^{422}\) (the default), construct **PWMOutputDevice** (page 142) instances for the motor controller pins, allowing both direction and variable speed control. If False\(^{423}\), construct **DigitalOutputDevice** (page 141) instances, allowing only direction control.

• **pin_factory** *(Factory (page 226) or None)* – See *API - Pins* (page 221) for more information (this is an advanced feature which most users can ignore).

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

Drive the motor backwards.

**Parameters**

**speed** *(float\(^{424}\))* – 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\(^{425}\) when the class was constructed (and only 0 or 1 if not).

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

Drive the motor forwards.

**Parameters**

**speed** *(float\(^{426}\))* – 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\(^{427}\) 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.

**property is_active**

Returns True\(^{428}\) if the motor is currently running and False\(^{429}\) otherwise.

**property 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.

\(^{411}\) https://en.wikipedia.org/wiki/H_bridge

\(^{412}\) https://en.wikipedia.org/wiki/H_bridge

\(^{413}\) https://docs.python.org/3.9/library/functions.html#int

\(^{414}\) https://docs.python.org/3.9/library/functions.html#str

\(^{415}\) https://docs.python.org/3.9/library/constants.html#None

\(^{416}\) https://docs.python.org/3.9/library/functions.html#int

\(^{417}\) https://docs.python.org/3.9/library/functions.html#str

\(^{418}\) https://docs.python.org/3.9/library/constants.html#None

\(^{419}\) https://docs.python.org/3.9/library/functions.html#int

\(^{420}\) https://docs.python.org/3.9/library/functions.html#str

\(^{421}\) https://docs.python.org/3.9/library/constants.html#None

\(^{422}\) https://docs.python.org/3.9/library/functions.html#bool

\(^{423}\) https://docs.python.org/3.9/library/constants.html#True

\(^{424}\) https://docs.python.org/3.9/library/constants.html#False

\(^{425}\) https://docs.python.org/3.9/library/constants.html#int

\(^{426}\) https://docs.python.org/3.9/library/constants.html#float

\(^{427}\) https://docs.python.org/3.9/library/constants.html#int

\(^{428}\) https://docs.python.org/3.9/library/constants.html#float

\(^{429}\) https://docs.python.org/3.9/library/constants.html#True
15.1.7 PhaseEnableMotor

class gpiozero.PhaseEnableMotor(*args, **kwargs)

Extends CompositeDevice (page 187) 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 241) 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 241) will be raised.

- **pwm** (bool) – If True (the default), construct PWMOutputDevice (page 142) instances for the motor controller pins, allowing both direction and variable speed control. If False, construct DigitalOutputDevice (page 141) instances, allowing only direction control.

- **pin_factory** (Factory or None) – See API - Pins (page 221) 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.

property is_active

Returns True if the motor is currently running and False otherwise.

property 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(*args, **kwargs)

Extends CompositeDevice (page 187) and represents a PWM-controlled servo motor connected to a GPIO pin.

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

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

```python
from gpiozero import Servo
from time import sleep

servo = Servo(17)

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

You can also use the `value` (page 138) property to move the servo to a particular position, on a scale from -1 (min) to 1 (max) where 0 is the mid-point:

```python
from gpiozero import Servo

servo = Servo(17)

servo.value = 0.5
```

**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 223) 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 241) 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 138)).

- **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 226) or **None**) - See API - Pins (page 221) 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 138) 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.

**property frame_width**
The time between control pulses, measured in seconds.

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

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

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

**property pulse_width**
Returns the current pulse width controlling the servo.

**property 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.

---

443 [https://docs.python.org/3.9/library/functions.html#int](https://docs.python.org/3.9/library/functions.html#int)
444 [https://docs.python.org/3.9/library/stdtypes.html#str](https://docs.python.org/3.9/library/stdtypes.html#str)
445 [https://docs.python.org/3.9/library/consts.html#None](https://docs.python.org/3.9/library/consts.html#None)
446 [https://docs.python.org/3.9/library/functions.html#float](https://docs.python.org/3.9/library/functions.html#float)
447 [https://docs.python.org/3.9/library/consts.html#None](https://docs.python.org/3.9/library/consts.html#None)
448 [https://docs.python.org/3.9/library/functions.html#float](https://docs.python.org/3.9/library/functions.html#float)
449 [https://docs.python.org/3.9/library/functions.html#float](https://docs.python.org/3.9/library/functions.html#float)
450 [https://docs.python.org/3.9/library/consts.html#None](https://docs.python.org/3.9/library/consts.html#None)
451 [https://docs.python.org/3.9/library/consts.html#None](https://docs.python.org/3.9/library/consts.html#None)
15.1.9 AngularServo

```python
class gpiozero.AngularServo(*args, **kwargs)
```

Extends Servo (page 137) 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 137) 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 139) 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 = 15
15.0
```

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

### Parameters

- **pin** (`int` 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 241) will be raised.

- **initial_angle** (`float`) – Sets the servo’s initial angle to the specified value. The default is 0. The value specified must be between `min_angle` and `max_angle` inclusive. `None` means to start the servo un-controlled (see `value` (page 140)).

- **min_angle** (`float`) – Sets the minimum angle that the servo can rotate to. This defaults to -90, but should be set to whatever you measure from your servo during calibration.

- **max_angle** (`float`) – Sets the maximum angle that the servo can rotate to. This defaults to 90, but should be set to whatever you measure from your servo during calibration.

- **min_pulse_width** (`float`) – The pulse width corresponding to the servo’s minimum position. This defaults to 1ms.

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

- **frame_width** (`float`) – The length of time between servo control pulses measured in seconds. This defaults to 20ms which is a common value for servos.
`pin_factory` ([Factory](page 226) or None) – See [API - Pins](page 221) for more information (this is an advanced feature which most users can ignore).

```python
max()
```

Set the servo to its maximum position.

```python
mid()
```

Set the servo to its mid-point position.

```python
min()
```

Set the servo to its minimum position.

**property angle**

The position of the servo as an angle measured in degrees. This will only be accurate if `min_angle` (page 140) and `max_angle` (page 140) have been set appropriately in the constructor.

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

**property is_active**

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

**property max_angle**

The maximum angle that the servo will rotate to when `max()` (page 140) is called.

**property min_angle**

The minimum angle that the servo will rotate to when `min()` (page 140) is called.

**property 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):

---

453 https://docs.python.org/3.9/library/functions.html#int
454 https://docs.python.org/3.9/library/stdtypes.html#str
455 https://docs.python.org/3.9/library/constants.html#None
456 https://docs.python.org/3.9/library/functions.html#float
457 https://docs.python.org/3.9/library/constants.html#None
458 https://docs.python.org/3.9/library/functions.html#float
459 https://docs.python.org/3.9/library/functions.html#float
460 https://docs.python.org/3.9/library/functions.html#float
461 https://docs.python.org/3.9/library/functions.html#float
462 https://docs.python.org/3.9/library/constants.html#None
463 https://docs.python.org/3.9/library/constants.html#None
464 https://docs.python.org/3.9/library/constants.html#None

---
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(*args, **kwargs)

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

This class extends OutputDevice (page 144) with a `blink()` (page 141) 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 241) will be raised.

- `active_high` *(bool)* – If True (the default), the on() (page 142) method will set the GPIO to HIGH. If False, the on() (page 142) method will set the GPIO to LOW (the off() (page 142) method always does the opposite).

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

- `pin_factory` *(Factory or None)* – See API - Pins (page 221) 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).

```python
off()
    Turns the device off.

on()
    Turns the device on.

property 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(*args, **kwargs)`

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

**Parameters**

• **pin (int or str)** – The GPIO pin that the device is connected to. See *Pin Numbering* (page 3) for valid pin numbers. If this is **None**, a **GPIODeviceError** (page 241) will be raised.

• **active_high (bool)** – If **True** (the default), the `on()` (page 143) method will set the GPIO to HIGH. If **False**, the `on()` (page 143) method will set the GPIO to LOW (the `off()` (page 143) 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 221) 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.

---

465 https://docs.python.org/3.9/library/functions.html#int
466 https://docs.python.org/3.9/library/stdtypes.html#str
467 https://docs.python.org/3.9/library/constants.html#None
468 https://docs.python.org/3.9/library/functions.html#bool
469 https://docs.python.org/3.9/library/constants.html#True
470 https://docs.python.org/3.9/library/constants.html#False
471 https://docs.python.org/3.9/library/functions.html#bool
472 https://docs.python.org/3.9/library/constants.html#False
473 https://docs.python.org/3.9/library/constants.html#None
474 https://docs.python.org/3.9/library/constants.html#True
475 https://docs.python.org/3.9/library/functions.html#float
476 https://docs.python.org/3.9/library/functions.html#float
477 https://docs.python.org/3.9/library/functions.html#int
478 https://docs.python.org/3.9/library/constants.html#None
479 https://docs.python.org/3.9/library/constants.html#True
480 https://docs.python.org/3.9/library/functions.html#bool
481 https://docs.python.org/3.9/library/constants.html#False
• **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 143) is 0.0), this changes it to “fully” on (**value** (page 143) is 1.0). If the device has a duty cycle (**value** (page 143)) of 0.1, this will toggle it to 0.9, and so on.

**property frequency**

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

**property is_active**

Returns True if the device is currently active (**value** (page 143) is non-zero) and False otherwise.

**property 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 gpiozero.OutputDevice(*args, **kwargs)

Represents a generic GPIO output device.

This class extends GPIODevice (page 122) to add facilities common to GPIO output devices: an on() (page 144) method to switch the device on, a corresponding off() (page 144) method, and a toggle() (page 144) 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 241) will be raised.

- **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 or None) – See API - Pins (page 221) 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.

property active_high

When True, the value (page 145) property is True when the device’s pin (page 123) is high.
When `False` the `value` (page 145) property is `True` when the device’s pin is low (i.e. the value is inverted).

This property can be set after construction; be warned that changing it will invert `value` (page 145) (i.e. changing this property doesn’t change the device’s pin state - it just changes how that state is interpreted).

**property 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 (*args, **kwargs)`

Extends `Device` (page 199). 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 123)).

**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 241) will be raised. If the pin is already in use by another device, `GPIOPinInUse` (page 241) 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 199) descendents can also be used as context managers using the `with` statement. For example:

```
```

---

519 https://docs.python.org/3.9/library/constants.html#True
520 https://docs.python.org/3.9/library/constants.html#True
521 https://docs.python.org/3.9/library/constants.html#True
522 https://docs.python.org/3.9/library/constants.html#False
523 https://docs.python.org/3.9/library/constants.html#int
524 https://docs.python.org/3.9/library/constants.html#str
525 https://docs.python.org/3.9/library/constants.html#None
526 https://docs.python.org/3.9/library/constants.html#False
527 https://docs.python.org/3.9/library/constants.html#None
528 https://docs.python.org/3.9/library/constants.html#True
529 https://docs.python.org/3.9/library/constants.html#False
530 https://docs.python.org/3.9/library/constants.html#False
531 https://docs.python.org/3.9/library/constants.html#True
532 https://docs.python.org/3.9/library/constants.html#True
533 https://docs.python.org/3.9/library/constants.html#True
534 https://docs.python.org/3.9/library/constants.html#True
property closed

Returns True\textsuperscript{527} if the device is closed (see the close() (page 122) method). Once a device is closed you can no longer use any other methods or properties to control or query the device.

property pin

The Pin (page 227) that the device is connected to. This will be None\textsuperscript{528} if the device has been closed (see the close() (page 199) method). When dealing with GPIO pins, query pin.number to discover the GPIO pin (in BCM numbering) that the device is connected to.

property 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.

\textsuperscript{523} https://docs.python.org/3.9/library/functions.html#int
\textsuperscript{524} https://docs.python.org/3.9/library/stdtypes.html#str
\textsuperscript{525} https://docs.python.org/3.9/library/constants.html#None
\textsuperscript{526} https://docs.python.org/3.9/reference/compound_stmts.html#with
\textsuperscript{527} https://docs.python.org/3.9/library/constants.html#True
\textsuperscript{528} https://docs.python.org/3.9/library/constants.html#None
API - SPI DEVICES

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

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

The gpiozero library provides two SPI implementations:

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

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).

You cannot mix these two schemes, i.e. attempting to specify `port` and `clock_pin` will result in `SPIBadArgs` (page 240) 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 221) for more information).

Hence the following constructors are all equivalent:

529 https://en.wikipedia.org/wiki/Serial_Peripheral_Interface_Bus
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

class gpiozero.MCP3001(*args, **kwargs)

The MCP3001 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-.

@property 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

class gpiozero.MCP3002(*args, **kwargs)

The MCP3002 is a 10-bit analog to digital converter with 2 channels (0-1).

@property 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.

@property 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 149) in differential mode, channel 0 is read relative to channel 1).

@property 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(*args, **kwargs)

The MCP3004\(^{532}\) is a 10-bit analog to digital converter with 4 channels (0-3).

property 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.

property differential

If \texttt{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 149) in differential mode, channel 0 is read relative to channel 1).

property 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(*args, **kwargs)

The MCP3008\(^{533}\) is a 10-bit analog to digital converter with 8 channels (0-7).

property 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.

property differential

If \texttt{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 149) in differential mode, channel 0 is read relative to channel 1).

property 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).

16.2.5 MCP3201

class gpiozero.MCP3201(*args, **kwargs)

The MCP3201\(^{534}\) 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-.

property 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).

\(^{532}\) http://www.farnell.com/datasheets/808965.pdf

\(^{533}\) http://www.farnell.com/datasheets/808965.pdf

\(^{534}\) http://www.farnell.com/datasheets/1669366.pdf
16.2.6 MCP3202

class gpiozero.MCP3202(*args, **kwargs)
The MCP3202\(^{535}\) is a 12-bit analog to digital converter with 2 channels (0-1).

property 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.

property 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 149) in differential mode, channel 0 is read relative to channel 1).

property 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(*args, **kwargs)
The MCP3204\(^{536}\) is a 12-bit analog to digital converter with 4 channels (0-3).

property 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.

property 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 149) in differential mode, channel 0 is read relative to channel 1).

property 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

class gpiozero.MCP3208(*args, **kwargs)
The MCP3208\(^{537}\) is a 12-bit analog to digital converter with 8 channels (0-7).

property 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.

\(^{535}\) http://www.farnell.com/datasheets/1669376.pdf
\(^{536}\) http://www.farnell.com/datasheets/808967.pdf
\(^{537}\) http://www.farnell.com/datasheets/1669376.pdf
**property 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 149) in differential mode, channel 0 is read relative to channel 1).

**property 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

class `gpiozero.MCP3301(*args, **kwargs)`

The `MCP3301`\(^538\) 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.

**property 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

class `gpiozero.MCP3302(*args, **kwargs)`

The `MCP3302`\(^539\) 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.

**property 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.

**property 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 152) in differential mode, channel 0 is read relative to channel 1).

**property 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 (*args, **kwargs)
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.

property 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.

property 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 152) in differential mode, channel 0 is read relative to channel 1).

property 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):

\[ http://www.farnell.com/datasheets/1486116.pdf \]
The following sections document these base classes for advanced users that wish to construct classes for their own devices.

### 16.3.1 AnalogInputDevice

```python
class gpiozero.AnalogInputDevice(*args, **kwargs)
```

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

Typical analog input devices are **analog to digital converters** (ADCs). Several classes are provided for specific ADC chips, including `MCP3004` (page 149), `MCP3008` (page 149), `MCP3204` (page 150), and `MCP3208` (page 150).

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

```python
from gpiozero import MCP3008

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

The `value` attribute is normalized such that its value is always between 0.0 and 1.0 (or in special cases, such as differential sampling, -1 to +1). Hence, you can use an analog input to control the brightness of
from gpiozero import MCP3008, PWMLED

c = MCP3008(0)
d = PWMLED(17)
d.source = c

The `voltage` (page 154) attribute reports values between 0.0 and `max_voltage` (which defaults to 3.3, the logic level of the GPIO pins).

**property bits**
The bit-resolution of the device/channel.

**property max_voltage**
The voltage required to set the device’s value to 1.

**property raw_value**
The raw value as read from the device.

**property 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).

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

### 16.3.2 SPIDevice

**class gpiozero.SPIDevice(*args, **kwargs)**

Extends `Device` (page 199). Represents a device that communicates via the SPI protocol.

See [SPI keyword args](#) for information on the keyword arguments that can be specified with the constructor.

**close()**

Shut down the device and release all associated resources (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 199) descendents can also be used as context managers using the `with` statement. For example:

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

**property closed**

Returns `True`\(^{543}\) if the device is closed (see the `close()` (page 154) method). Once a device is closed you can no longer use any other methods or properties to control or query the device.

\(^{542}\) https://docs.python.org/3.9/reference/compound_stmts.html#with
\(^{543}\) https://docs.python.org/3.9/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 105) and API - Output Devices (page 125) 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(*args, **kwargs)

    Extends LEDCollection (page 186) 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:

- *pills* – 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 157) instances to create trees of LEDs.

- pwm (bool) – If True, construct PWMLED (page 127) instances for each pin. If False (the default), construct regular LED (page 125) instances.

- active_high (bool) – If True (the default), the on () (page 158) method will set all the associated pins to HIGH. If False, the on () (page 158) method will set all pins to LOW (the off() (page 158) 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\(\text{list}^{554} \text{ or } \text{None}\) – 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.

• \text{pin\_factory}\(\text{Factory}^{(\text{page 226})} \text{ or } \text{None}\) – See API - Pins (page 221) for more information (this is an advanced feature which most users can ignore).

• \**\text{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 157) instances to create trees of LEDs.

\text{blink}(\text{on\_time}=1, \text{off\_time}=1, \text{fade\_in\_time}=0, \text{fade\_out\_time}=0, n=\text{None}, \text{background}=\text{True})

Make all the LEDs turn on and off repeatedly.

Parameters

• \text{on\_time}(\text{float}^{555}) – Number of seconds on. Defaults to 1 second.

• \text{off\_time}(\text{float}^{556}) – Number of seconds off. Defaults to 1 second.

• \text{fade\_in\_time}(\text{float}^{557}) – Number of seconds to spend fading in. Defaults to 0. Must be 0 if \text{pwm}^{558} was \text{False}^{558} when the class was constructed (\text{ValueError}^{559} will be raised if not).

• \text{fade\_out\_time}(\text{float}^{560}) – Number of seconds to spend fading out. Defaults to 0. Must be 0 if \text{pwm}^{560} was \text{False}^{560} when the class was constructed (\text{ValueError}^{561} will be raised if not).

• \text{n}(\text{int}^{561} \text{ or } \text{None}) – Number of times to blink; \text{None}^{564} (the default) means forever.

• \text{background}(\text{bool}^{565}) – If \text{True}^{566}, start a background thread to continue blinking and return immediately. If \text{False}^{567}, only return when the blink is finished (warning: the default value of \text{n} will result in this method never returning).

\text{off}(\text{*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() # 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 \text{blink()} (page 158) is currently active, it will be stopped first.

Parameters

• \text{args}(\text{int}^{568}) – The index(es) of the LED(s) to turn off. If no indexes are specified turn off all LEDs.

\text{on}(\text{*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)
```
If blink() (page 158) 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.

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

**toggle** (*args)

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

```python
from gpiozero import LEDBoard

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

If blink() (page 158) is currently active, it will be stopped first.

**Parameters**

- **args** (int) – 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(*args, **kwargs)

Extends LEDCollection (page 186) 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 161) and values (page 161) are supported:

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

graph = LEDBarGraph(2, 3, 4, 5, 6, pwm=True)
pot = MCP3008(channel=0)

graph.source = pot
pause()
```
Parameters

- **pins** – Specify the GPIO pins that the LEDs of the bar graph are attached to. See Pin Numbering (page 3) for valid pin numbers. You can designate as many pins as necessary.

- **pwm** (bool) – If True, construct PWMLED (page 127) instances for each pin. If False (the default), construct regular LED (page 125) instances. This parameter can only be specified as a keyword parameter.

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

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

- **pin_factory** (Factory or None) – See API - Pins (page 221) for more information (this is an advanced feature which most users can ignore).

**property lit_count**

The number of LEDs on the bar graph actually lit up. Note that just like value (page 161), this can be negative if the LEDs are lit from last to first.

**property source**

The iterable to use as a source of values for value (page 161).

**property 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 161) is effectively -1 < value <= 1.

**property values**

An infinite iterator of values read from value (page 161).

---

579 https://docs.python.org/3.9/library/functions.html#bool
580 https://docs.python.org/3.9/library/constants.html#True
581 https://docs.python.org/3.9/library/constants.html#False
582 https://docs.python.org/3.9/library/functions.html#bool
583 https://docs.python.org/3.9/library/constants.html#True
584 https://docs.python.org/3.9/library/constants.html#False
585 https://docs.python.org/3.9/library/functions.html#float
17.1.3 LEDCharDisplay

```python
class gpiozero.LEDCharDisplay(*args, **kwargs)
```
Extends `LEDCollection` (page 186) for a multi-segment LED display.

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

```
a
f  b
g  c
e  d  dp
```

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

```
a
f  j  i  b
\  \ /  \  k
```

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 163) 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 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` (page 163) 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` (page 127) instances for each pin. If `False` (the default), construct regular `LED` (page 125) instances.

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

• `initial_value` – 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 226) for more information (this is an advanced feature which most users can ignore).

property font

An `LEDCharFont` (page 165) 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` (page 163) never alters the underlying LED states. Only assignment to `value` (page 163), or calling the inherited `LEDCollection` (page 186) methods (`on()`, `off()`, etc.) modifies LED states. However, modifying the font may alter the character returned by querying `value` (page 163).

property value

The character the display should show. This is mapped by the current `font` (page 163) 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` (page 163), the value is `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 gpiozero.LEDMultiCharDisplay (*args, **kwargs)

Wraps `LEDCharDisplay` (page 162) for multi-character multiplexed LED character displays.

The class is constructed with a `char` which is an instance of the `LEDCharDisplay` (page 162) class, capable of controlling the LEDs in one character of the display, and an additional set of `pins` that represent the common cathode (or anode) of each character.

---

586 https://en.wikipedia.org/wiki/Seven-segment_display
587 https://docs.python.org/3.9/library/constats.html#False
588 https://docs.python.org/3.9/library/stdtypes.html#str
589 https://docs.python.org/3.9/library/stdtypes.html#dict
590 https://docs.python.org/3.9/library/stdtypes.html#str
591 https://docs.python.org/3.9/library/functions.html#int
592 https://docs.python.org/3.9/library/stdtypes.html#str
593 https://docs.python.org/3.9/library/stdtypes.html#dict
594 https://docs.python.org/3.9/library/functions.html#bool
595 https://docs.python.org/3.9/library/constats.html#True
596 https://docs.python.org/3.9/library/constats.html#False
597 https://docs.python.org/3.9/library/functions.html#bool
598 https://docs.python.org/3.9/library/constats.html#True
599 https://docs.python.org/3.9/library/constats.html#False
600 https://docs.python.org/3.9/library/constats.html#None
601 https://docs.python.org/3.9/library/constats.html#None
**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 `pins`.

The `active_high` parameter defaults to `True`. Note that it only applies to the specified `pins`, which are assumed to be controlling a set of transistors (hence the default). The specified `char` will use its own `active_high` parameter. Finally, `initial_value` defaults to a tuple of `value` (page 163) attribute of the specified display multiplied by the number of `pins` provided.

When the `value` (page 164) 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 to display the complete value.

**property 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.

**property value**

The sequence of values to display.

This can be any sequence containing keys from the `font` (page 163) 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):

```python
from gpiozero import LEDCharDisplay, LEDMultiCharDisplay

c = LEDCharDisplay(4, 5, 6, 7, 8, 9, 10)
d = LEDMultiCharDisplay(c, 19, 20, 21, 22)
d.value = 'LEDS'
```

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:

```python
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.

---

602 [https://en.wikipedia.org/wiki/Multiplexed_display](https://en.wikipedia.org/wiki/Multiplexed_display)
603 [https://docs.python.org/3.9/library/constants.html#True](https://docs.python.org/3.9/library/constants.html#True)
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 162), 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\footnote{https://docs.python.org/3.9/library/stdtypes.html#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, 0, 1, 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 162). However, changing the font will not change the state of the LEDs in the display (though it may
change the value (page 163) of the display when next queried).

\textbf{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 162).

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 212).

17.1.6 ButtonBoard

class gpiozero.ButtonBoard(*args, **kwargs)

Extends CompositeDevice (page 187) and represents a generic button board or collection of buttons. The
value (page 166) 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 157) with a ButtonBoard (page 165):

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

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

pause()
```

Alternatively you could represent the number of pressed buttons with an LEDBarGraph (page 160):

```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)
```

(continues on next page)
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 121) for more information.

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

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

- **hold_repeat** (**bool**) – If True, the **when_held** handler will be repeatedly executed as long as any buttons remain held, every **hold_time** seconds. If False (the default) the **when_held** handler will be only be executed once per hold.

- **pin_factory** (**Factory** (page 226) or **None**) – See API - Pins (page 221) for more information (this is an advanced feature which most users can ignore).

- ****named_pins** – Specify GPIO pins that buttons of the board are attached to, associating each button with a property name. You can designate as many pins as necessary and use any names, provided they’re not already in use by something else.

**wait_for_press** *(timeout=None)*

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

Parameters

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

**wait_for_release** *(timeout=None)*

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

Parameters

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

**property is_pressed**

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

**property pressed_time**

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

**property value**

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

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

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

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

**when_released**

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

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

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

### 17.1.7 TrafficLights

```python
class gpiozero.TrafficLights(*args, **kwargs)
    Extends LEDBoard (page 157) 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 :class:`PWMLED` (page 127) instances to represent each LED. If False (the default), construct regular :class:`LED` (page 125) 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** (:class:`Factory` (page 226) or *None*) – See API - Pins (page 221) for more information (this is an advanced feature which most users can ignore).

  red

  The red :class:`LED` (page 125) or :class:`PWMLED` (page 127).

  amber

  The amber :class:`LED` (page 125) or :class:`PWMLED` (page 127). Note that this attribute will not be present when the instance is constructed with the *yellow* keyword parameter.

  yellow

  The yellow :class:`LED` (page 125) or :class:`PWMLED` (page 127). Note that this attribute will only be present when the instance is constructed with the *yellow* keyword parameter.

  green

  The green :class:`LED` (page 125) or :class:`PWMLED` (page 127).

### 17.1.8 TrafficLightsBuzzer

**class** `gpiozero.TrafficLightsBuzzer (*args, **kwargs)`

Extends :class:`CompositeOutputDevice` (page 186) and is a generic class for HATs with traffic lights, a button and a buzzer.

**Parameters**

• **lights** (:class:`TrafficLights` (page 167)) – An instance of :class:`TrafficLights` (page 167) representing the traffic lights of the HAT.

• **buzzer** (:class:`Buzzer` (page 131)) – An instance of :class:`Buzzer` (page 131) representing the buzzer on the HAT.

• **button** (:class:`Button` (page 105)) – An instance of :class:`Button` (page 105) representing the button on the HAT.

• **pin_factory** (:class:`Factory` (page 226) or *None*) – See API - Pins (page 221) for more information (this is an advanced feature which most users can ignore).

**lights**

The :class:`TrafficLights` (page 167) instance passed as the *lights* parameter.
buzzer
The **Buzzer** (page 131) instance passed as the **buzzer** parameter.

button
The **Button** (page 105) instance passed as the **button** parameter.

17.1.9 PiHutXmasTree

**class gpiozero.PiHutXmasTree(*args, **kwargs)**

Extends **LEDBoard** (page 157) for the **Pi Hut's Xmas board**[640]: 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 169) attribute. Alternatively, as with all descendents of **LEDBoard** (page 157), you can treat the instance as a sequence of LEDs (the first element is the **star** (page 169)).

The Xmas Tree board pins are fixed and therefore there’s no need to specify them when constructing this class. The following example turns all the LEDs on one at a time:

```python
from gpiozero import PiHutXmasTree
from time import sleep

tree = PiHutXmasTree()

for light in tree:
    light.on()
    sleep(1)
```

The following example turns the star LED on and sets all the red LEDs to flicker randomly:

```python
from gpiozero import PiHutXmasTree
from gpiozero.tools import random_values
from signal import pause

tree = PiHutXmasTree(pwm=True)

tree.star.on()

for led in tree[1:]:
    led.source_delay = 0.1
    led.source = random_values()

pause()
```

**Parameters**

- **pwm** ([bool](641)) – If **True**[642], construct **PWMLED** (page 127) instances for each pin. If **False**[643] (the default), construct regular **LED** (page 125) instances.

- **initial_value** ([bool](644) or **None**) – If **False**[645] (the default), all LEDs will be off initially. If **None**[646], each device will be left in whatever state the pin is found in when configured for output (warning: this can be on). If **True**[647], the device will be switched on initially.

- **pin_factory** ([Factory](page 226) or **None**) – See **API - Pins** (page 221) for more information (this is an advanced feature which most users can ignore).

**star**

Returns the **LED** (page 125) or **PWMLED** (page 127) representing the white star on top of the tree.
led0, led1, led2, ...

Returns the LED (page 125) or PWMLED (page 127) 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(*args, **kwargs)

Extends RGBLED (page 128) for the PiBorg LedBorg648: 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
from gpiozero import LedBorg

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

Parameters

- **initial_value** (Color649 or tuple650) – The initial color for the LedBorg. Defaults to black (0, 0, 0).
- **pwm** (bool651) – If True652 (the default), construct PWMLED (page 127) instances for each component of the LedBorg. If False653, construct regular LED (page 125) instances, which prevents smooth color gradations.
- **pin_factory** (Factory page 226 or None) – See API - Pins (page 221) for more information (this is an advanced feature which most users can ignore).

17.1.11 PiLiter

class gpiozero.PiLiter(*args, **kwargs)

Extends LEDBoard (page 157) for the Ciseco Pi-LITEr654: 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 127) instances for each pin. If `False` (the default), construct regular `LED` (page 125) 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`, the each LED will be switched on initially.

• **pin_factory** *(Factory or None)* – See API - Pins (page 221) for more information (this is an advanced feature which most users can ignore).

### 17.1.12 PiLiterBarGraph

class `gpiozero.PiLiterBarGraph(*args, **kwargs)`

Extends `LEDBarGraph` (page 160) to treat the Ciseco Pi-LITEr as an 8-segment bar graph.

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

```python
from gpiozero import PiLiterBarGraph

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

**Parameters**

• **pwm** *(bool)* – If `True`, construct `PWMLED` (page 127) instances for each pin. If `False` (the default), construct regular `LED` (page 125) instances.

• **initial_value** *(float)* – The initial value of the graph given as a float between -1 and +1. Defaults to 0.0.

• **pin_factory** *(Factory or None)* – See API - Pins (page 221) for more information (this is an advanced feature which most users can ignore).

### 17.1.13 PiTraffic

class `gpiozero.PiTraffic(*args, **kwargs)`

Extends `TrafficLights` (page 167) for the Low Voltage Labs PI-TRAFFIC vertical traffic lights board when attached to GPIO pins 9, 10, and 11.

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

```python
http://shop.ciseco.co.uk/pi-liter-8-led-strip-for-the-raspberry-pi/
https://docs.python.org/3.9/library/functions.html#bool
https://docs.python.org/3.9/library/constants.html#True
https://docs.python.org/3.9/library/constants.html#False
https://docs.python.org/3.9/library/functions.html#bool
https://docs.python.org/3.9/library/constants.html#False
https://docs.python.org/3.9/library/constants.html#None
https://docs.python.org/3.9/library/constants.html#True
http://shop.ciseco.co.uk/pi-liter-8-led-strip-for-the-raspberry-pi/
https://docs.python.org/3.9/library/functions.html#bool
https://docs.python.org/3.9/library/constants.html#True
https://docs.python.org/3.9/library/constants.html#False
https://docs.python.org/3.9/library/constants.html#None
https://docs.python.org/3.9/library/constants.html#False
http://shop.ciseco.co.uk/pi-liter-8-led-strip-for-the-raspberry-pi/
To use the PI-TRAFFIC board when attached to a non-standard set of pins, simply use the parent class, `TrafficLights` (page 167).

**Parameters**

- `pwm` *(bool)* – If `True`, construct `PWMLED` (page 127) instances to represent each LED. If `False` (the default), construct regular `LED` (page 125) 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 226) or None)* – See `API - Pins` (page 221) for more information (this is an advanced feature which most users can ignore).

### 17.1.14 PiStop

**class** `gpiozero.PiStop(*args, **kwargs)`

Extends `TrafficLights` (page 167) 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 127) instances to represent each LED. If `False` (the default), construct regular `LED` (page 125) 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 226) or None)* – See `API - Pins` (page 221) for more information (this is an advanced feature which most users can ignore).

667 http://lowvoltagelabs.com/products/pi-traffic/
668 https://docs.python.org/3.9/library/functions.html#bool
669 https://docs.python.org/3.9/library/constants.html#True
670 https://docs.python.org/3.9/library/constants.html#False
671 https://docs.python.org/3.9/library/functions.html#bool
672 https://docs.python.org/3.9/library/constants.html#False
673 https://docs.python.org/3.9/library/constants.html#None
674 https://docs.python.org/3.9/library/constants.html#True
17.1.15 FishDish

```python
class gpiozero.FishDish(*args, **kwargs):
    Extends CompositeOutputDevice (page 186) for the Pi Supply FishDish: traffic light LEDs, a button and a buzzer.
    The FishDish pins are fixed and therefore there's no need to specify them when constructing this class. The following example waits for the button to be pressed on the FishDish, then turns on all the LEDs:
```  
```python
from gpiozero import FishDish
fish = FishDish()
fish.button.wait_for_press()
fish.lights.on()
```

Parameters

- `pwm` (bool) – If True, construct PWMLED (page 127) instances to represent each LED. If False (the default), construct regular LED (page 125) instances.
- `pin_factory` (Factory or None) – See API - Pins (page 221) for more information (this is an advanced feature which most users can ignore).

17.1.16 TrafficHat

```python
class gpiozero.TrafficHat(*args, **kwargs):
    Extends CompositeOutputDevice (page 186) for the Pi Supply Traffic HAT: a board with traffic light LEDs, a button and a buzzer.
    The Traffic HAT pins are fixed and therefore there's no need to specify them when constructing this class. The following example waits for the button to be pressed on the Traffic HAT, then turns on all the LEDs:
```  
```python
from gpiozero import TrafficHat
hat = TrafficHat()
hat.button.wait_for_press()
hat.lights.on()
```

Parameters

- `pwm` (bool) – If True, construct PWMLED (page 127) instances to represent each LED. If False (the default), construct regular LED (page 125) instances.
- `pin_factory` (Factory or None) – See API - Pins (page 221) for more information (this is an advanced feature which most users can ignore).

---

675 https://pihw.wordpress.com/meltwaters-pi-hardware-kits/pi-stop/
676 https://docs.python.org/3.9/library/stdtypes.html#str
677 https://github.com/PiHw/Pi-Stop/blob/master/markdown_source/markdown/Discover-PiStop.md
678 https://docs.python.org/3.9/library/functions.html#bool
679 https://docs.python.org/3.9/library/constants.html#True
680 https://docs.python.org/3.9/library/constants.html#False
681 https://docs.python.org/3.9/library/functions.html#bool
682 https://docs.python.org/3.9/library/constants.html#False
683 https://docs.python.org/3.9/library/constants.html#None
684 https://docs.python.org/3.9/library/constants.html#True
685 https://www.pi-supply.com/product/fish-dish-raspberry-pi-led-buzzer-board/
686 https://docs.python.org/3.9/library/functions.html#bool
687 https://docs.python.org/3.9/library/constants.html#True
688 https://docs.python.org/3.9/library/constants.html#False
17.1.17 TrafficpHat

class gpiozero.TrafficpHat(*args, **kwargs)

Extends TrafficLights (page 167) for the Pi Supply Traffic pHAT\(^\text{693}\): a small board with traffic light LEDs.

The Traffic pHAT pins are fixed and therefore there's no need to specify them when constructing this class. The following example then turns on all the LEDs:

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

**Parameters**

- `pwm (bool)` – If True\(^\text{695}\), construct PWMLED (page 127) instances to represent each LED. If False\(^\text{696}\) (the default), construct regular LED (page 125) instances.
- `initial_value (bool or None)` – If False\(^\text{698}\) (the default), all LEDs will be off initially. If None\(^\text{699}\), each device will be left in whatever state the pin is found in when configured for output (warning: this can be on). If True\(^\text{700}\), the device will be switched on initially.
- `pin_factory (Factory or None)` – See API - Pins (page 221) for more information (this is an advanced feature which most users can ignore).

17.1.18 JamHat

class gpiozero.JamHat(*args, **kwargs)

Extends CompositeOutputDevice (page 186) for the ModMyPi JamHat\(^\text{701}\) 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\(^\text{703}\), construct PWMLED (page 127) instances to represent each LED on the board. If False\(^\text{704}\) (the default), construct regular LED (page 125) instances.

---

\(^{693}\)[https://uk.pi-supply.com/products/traffic-hat-for-raspberry-pi]
\(^{694}\)[https://docs.python.org/3.9/library/functions.html#bool]
\(^{695}\)[https://docs.python.org/3.9/library/constants.html#True]
\(^{696}\)[https://docs.python.org/3.9/library/constants.html#False]
\(^{697}\)[http://pisupp.ly/trafficphat]
\(^{698}\)[https://docs.python.org/3.9/library/functions.html#bool]
\(^{699}\)[https://docs.python.org/3.9/library/constants.html#False]
\(^{700}\)[https://docs.python.org/3.9/library/constants.html#True]
\(^{701}\)[https://docs.python.org/3.9/library/constants.html#None]
\(^{702}\)[https://docs.python.org/3.9/library/constants.html#False]
\(^{703}\)[https://docs.python.org/3.9/library/constants.html#True]
\(^{704}\)[https://docs.python.org/3.9/library/constants.html#False]

Chapter 17. API - Boards and Accessories
*pin_factory*(Factory (page 226) or None) – See API - Pins (page 221) for more information (this is an advanced feature which most users can ignore).

`lights_1, lights_2`
Two `LEDBoard` (page 157) instances representing the top (`lights_1`) and bottom (`lights_2`) rows of LEDs on the JamHat.

`red, yellow, green`
`LED` (page 125) or `PWMLED` (page 127) 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 105) objects on the JamHat.

`buzzer`
The `TonalBuzzer` (page 133) 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(*args, **kwargs)`

Extends `CompositeOutputDevice` (page 186) 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()
pb.buzzer.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**

701 https://thepihut.com/products/jam-hat
702 https://docs.python.org/3.9/library/functions.html#bool
703 https://docs.python.org/3.9/library/constanst.html#True
704 https://docs.python.org/3.9/library/constanst.html#False
• `pwm` *(bool)* – If `True` (page 127), construct `PWMLED` (page 127) instances to represent each LED on the board, otherwise if `False` (the default), construct regular `LED` (page 125) instances.

• `pin_factory` *(Factory or None)* – See *API - Pins* (page 221) for more information (this is an advanced feature which most users can ignore).

```
lights
TrafficLights (page 167) instance representing the three LEDs
   red, amber, green
   LED (page 125) or PWMLED (page 127) instances representing the red, amber, and green LEDs

button
   The red Button (page 105) object on the Pibrella

buzzer
   A TonalBuzzer (page 133) object representing the buzzer

inputs
   A namedtuple() of the input pin numbers
   a, b, c, d

outputs
   A namedtuple() 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

```python
from gpiozero import Robot
robot = Robot(left=Motor(4, 14), right=Motor(17, 18))
robot.forward()
```

**Parameters**

• `left` *(Motor or PhaseEnableMotor)* – A `Motor` (page 134) or a `PhaseEnableMotor` (page 136) for the left wheel of the robot.

• `right` *(Motor or PhaseEnableMotor)* – A `Motor` (page 134) or a `PhaseEnableMotor` (page 136) for the right wheel of the robot.

---

705 http://www.pibrella.com/
706 https://docs.python.org/3.9/library/functions.html#bool
707 https://docs.python.org/3.9/library/constants.html#True
708 https://docs.python.org/3.9/library/constants.html#False
709 https://docs.python.org/3.9/library/collections.html#collections.namedtuple
710 https://docs.python.org/3.9/library/collections.html#collections.namedtuple

Chapter 17. API - Boards and Accessories
• **pin_factory** *(Factory (page 226) or None) – See API - Pins (page 221) for more information (this is an advanced feature which most users can ignore).

**left_motor**

The *Motor* (page 134) on the left of the robot.

**right_motor**

The *Motor* (page 134) on the right of the robot.

**backward** *(speed=1.0, curve_left=0, curve_right=0)*

Drive the robot backward by running both motors backward.

**Parameters**

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

**forward** *(speed=1.0, 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.

**left** *(speed=1)*

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

**Parameters**

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

**reverse** ()

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

**right** *(speed=1)*

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

**Parameters**

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

**property 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.

### 17.1.21 PhaseEnableRobot

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

Deprecated alias of `Robot` (page 176). The `Robot` (page 176) class can now be constructed directly with `Motor` (page 134) or `PhaseEnableMotor` (page 136) classes.

### 17.1.22 RyanteckRobot

**class** `gpiozero.RyanteckRobot` (*.args, **kwargs)*

Extends `Robot` (page 176) for the Ryanteck motor controller board.\(^{719}\)

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

```python
from gpiozero import RyanteckRobot
robot = RyanteckRobot()
robot.forward()
```

**Parameters**

- **pwm** *(bool)* — If `True` (the default), construct `PWMOutputDevice` (page 142) instances for the motor controller pins, allowing both direction and variable speed control. If `False`, construct `DigitalOutputDevice` (page 141) instances, allowing only direction control.

- **pin_factory** *(Factory or None)* — See `API - Pins` (page 221) for more information (this is an advanced feature which most users can ignore).

---

---

\(^{711}\) [https://docs.python.org/3.9/library/functions.html#float](https://docs.python.org/3.9/library/functions.html#float)

\(^{712}\) [https://docs.python.org/3.9/library/functions.html#float](https://docs.python.org/3.9/library/functions.html#float)

\(^{713}\) [https://docs.python.org/3.9/library/functions.html#float](https://docs.python.org/3.9/library/functions.html#float)

\(^{714}\) [https://docs.python.org/3.9/library/functions.html#float](https://docs.python.org/3.9/library/functions.html#float)

\(^{715}\) [https://docs.python.org/3.9/library/functions.html#float](https://docs.python.org/3.9/library/functions.html#float)

\(^{716}\) [https://docs.python.org/3.9/library/functions.html#float](https://docs.python.org/3.9/library/functions.html#float)

\(^{717}\) [https://docs.python.org/3.9/library/functions.html#float](https://docs.python.org/3.9/library/functions.html#float)

\(^{718}\) [https://docs.python.org/3.9/library/functions.html#float](https://docs.python.org/3.9/library/functions.html#float)


\(^{720}\) [https://docs.python.org/3.9/library/functions.html#bool](https://docs.python.org/3.9/library/functions.html#bool)

\(^{721}\) [https://docs.python.org/3.9/library/constan ts.html#True](https://docs.python.org/3.9/library/constants.html#True)

\(^{722}\) [https://docs.python.org/3.9/library/constan ts.html#False](https://docs.python.org/3.9/library/constants.html#False)
17.1.23 CamJamKitRobot

class gpiozero.CamJamKitRobot (*args, **kwargs)
    Extends Robot (page 176) for the CamJam #3 EduKit\(^{723}\) 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\(^{724}\)) – If True\(^{725}\) (the default), construct PWMOutputDevice (page 142)
  instances for the motor controller pins, allowing both direction and variable speed control.
  If False\(^{726}\), construct DigitalOutputDevice (page 141) instances, allowing only
direction control.

- `pin_factory` (Factory (page 226) or None) – See API - Pins (page 221) for
  more information (this is an advanced feature which most users can ignore).

17.1.24 PololuDRV8835Robot

class gpiozero.PololuDRV8835Robot (*args, **kwargs)
    Extends Robot (page 176) for the Pololu DRV8835 Dual Motor Driver Kit\(^{727}\).

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\(^{728}\)) – If True\(^{729}\) (the default), construct PWMOutputDevice (page 142)
  instances for the motor controller’s enable pins, allowing both direction and variable speed
  control. If False\(^{730}\), construct DigitalOutputDevice (page 141) instances, allowing
  only direction control.

- `pin_factory` (Factory (page 226) or None) – See API - Pins (page 221) for
  more information (this is an advanced feature which most users can ignore).

---

\(^{723}\) [http://camjam.me/?page_id=1035]

\(^{724}\) [https://docs.python.org/3.9/library/functions.html#bool]

\(^{725}\) [https://docs.python.org/3.9/library/constants.html#True]

\(^{726}\) [https://docs.python.org/3.9/library/constants.html#False]

\(^{727}\) [https://www.pololu.com/product/2753]

\(^{728}\) [https://docs.python.org/3.9/library/functions.html#bool]

\(^{729}\) [https://docs.python.org/3.9/library/constants.html#True]

\(^{730}\) [https://docs.python.org/3.9/library/constants.html#False]
17.1.25 Energenie

class gpiozero.Energenie(*args, **kwargs)

Extends Device (page 199) to represent an Energenie socket\(^1\) controller.

This class is constructed with a socket number and an optional initial state (defaults to False\(^2\), meaning off). Instances of this class can be used to switch peripherals on and off. For example:

```
from gpiozero import Energenie
lamp = Energenie(1)
lamp.on()
```

**Parameters**

- **socket** (int\(^3\)) – Which socket this instance should control. This is an integer number between 1 and 4.
- **initial_value** (bool\(^4\) or None) – 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\(^5\) which will switch the socket off. Specifying None\(^6\) will not set any initial state nor transmit any control signal to the device.
- **pin_factory** (Factory (page 226) or None) – See API - Pins (page 221) for more information (this is an advanced feature which most users can ignore).

**Methods**

- **off()**
  
  Turns the socket off.

- **on()**
  
  Turns the socket on.

**property** **socket**

Returns the socket number.

**property** **value**

Returns True\(^7\) if the socket is on and False\(^8\) if the socket is off. Setting this property changes the state of the socket. Returns None\(^9\) only when constructed with initial_value set to None\(^10\) and neither on() (page 180) nor off() (page 180) have been called since construction.

---

\(^1\) [https://energenie4u.co.uk/index.php/catalogue/product/ENER002-2PI](https://energenie4u.co.uk/index.php/catalogue/product/ENER002-2PI)

\(^2\) [https://docs.python.org/3.9/library/constants.html#False](https://docs.python.org/3.9/library/constants.html#False)

\(^3\) [https://docs.python.org/3.9/library/functions.html#int](https://docs.python.org/3.9/library/functions.html#int)

\(^4\) [https://docs.python.org/3.9/library/functions.html#bool](https://docs.python.org/3.9/library/functions.html#bool)

\(^5\) [https://docs.python.org/3.9/library/constants.html#False](https://docs.python.org/3.9/library/constants.html#False)

\(^6\) [https://docs.python.org/3.9/library/constants.html#None](https://docs.python.org/3.9/library/constants.html#None)

\(^7\) [https://docs.python.org/3.9/library/constants.html#True](https://docs.python.org/3.9/library/constants.html#True)

\(^8\) [https://docs.python.org/3.9/library/constants.html#False](https://docs.python.org/3.9/library/constants.html#False)

\(^9\) [https://docs.python.org/3.9/library/constants.html#None](https://docs.python.org/3.9/library/constants.html#None)

\(^10\) [https://docs.python.org/3.9/library/constants.html#None](https://docs.python.org/3.9/library/constants.html#None)
17.1.26 StatusZero

```python
class gpiozero.StatusZero(*args, **kwargs)

Extends LEDBoard (page 157) 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 125) 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 127) instances to represent each LED. If False (the default), construct regular LED (page 125) 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 226) or None) – See API - Pins (page 221) for more information (this is an advanced feature which most users can ignore).

your-label-here, your-label-here, ...

This entry represents one of the three labelled attributes supported on the STATUS Zero board. It is an LEDBoard (page 157) which contains:

- **red**

  The LED (page 125) or PWMLED (page 127) representing the red LED next to the label.

- **green**

  The LED (page 125) or PWMLED (page 127) representing the green LED next to the label.

---

741 https://thepihut.com/statuszero
742 https://docs.python.org/3.9/library/stdtypes.html#str
743 https://docs.python.org/3.9/library/functions.html#bool
744 https://docs.python.org/3.9/library/constants.html#True
745 https://docs.python.org/3.9/library/constants.html#False
746 https://docs.python.org/3.9/library/constants.html#None
747 https://docs.python.org/3.9/library/constants.html#True
748 https://docs.python.org/3.9/library/constants.html#False
749 https://docs.python.org/3.9/library/constants.html#None

17.1. Regular Classes 181
class gpiozero.StatusBoard(*args, **kwargs)

Extends CompositeOutputDevice (page 186) 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 157) containing two LED (page 125) objects named “red” and “green”, and a Button (page 105) 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 127) instances to represent each LED. If False (the default), construct regular LED (page 125) 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 221) for more information (this is an advanced feature which most users can ignore).

**your-label-here, your-label-here, ...**

This entry represents one of the five labelled attributes supported on the STATUS board. It is an CompositeOutputDevice (page 186) which contains:

- **lights**
  A LEDBoard (page 157) representing the lights next to the label. It contains:
  - **red**
    The LED (page 125) or PWMLED (page 127) representing the red LED next to the label.
  - **green**
    The LED (page 125) or PWMLED (page 127) representing the green LED next to the label.

- **button**
  A Button (page 105) representing the button next to the label.
17.1.28 SnowPi

```python
from gpiozero import SnowPi

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

Parameters

- `pwm (bool)`: If `True`, construct `PWMLED` (page 127) instances to represent each LED. If `False` (the default), construct regular `LED` (page 125) 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 221) for more information (this is an advanced feature which most users can ignore).

arms

A `LEDBoard` (page 157) representing the arms of the snow man. It contains the following attributes:

- **left**, **right**

  Two `LEDBoard` (page 157) objects representing the left and right arms of the snow-man. They contain:

  - **top**, **middle**, **bottom**

    The `LED` (page 125) or `PWMLED` (page 127) down the snow-man’s arms.

eyes

A `LEDBoard` (page 157) representing the eyes of the snow-man. It contains:

- **left**, **right**

  The `LED` (page 125) or `PWMLED` (page 127) for the snow-man’s eyes.

nose

The `LED` (page 125) or `PWMLED` (page 127) for the snow-man’s nose.

---

759 https://ryanteck.uk/raspberry-pi/114-snowpi-the-gpio-snowman-for-raspberry-pi-0635648608303.html
760 https://docs.python.org/3.9/library/functions.html#bool
761 https://docs.python.org/3.9/library/constants.html#True
762 https://docs.python.org/3.9/library/constants.html#False
763 https://docs.python.org/3.9/library/functions.html#bool
764 https://docs.python.org/3.9/library/constants.html#False
765 https://docs.python.org/3.9/library/constants.html#None
766 https://docs.python.org/3.9/library/constants.html#True
17.1.29 PumpkinPi

class gpiozero.PumpkinPi(*args, **kwargs)

Extends LEDBoard (page 157) for the ModMyPi PumpkinPi 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, construct PWMLED (page 127) instances to represent each LED. If False (the default), construct regular LED (page 125) 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 221) for more information (this is an advanced feature which most users can ignore).

sides

A LEDBoard (page 157) representing the LEDs around the edge of the pumpkin. It contains:

left, right

Two LEDBoard (page 157) instances representing the LEDs on the left and right sides of the pumpkin. They each contain:

top, midtop, middle, midbottom, bottom

Each LED (page 125) or PWMLED (page 127) around the specified side of the pumpkin.

eyes

A LEDBoard (page 157) representing the eyes of the pumpkin. It contains:

left, right

The LED (page 125) or PWMLED (page 127) 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:

767 https://www.modmypi.com/halloween-pumpkin-programmable-kit
768 https://docs.python.org/3.9/library/functions.html#bool
769 https://docs.python.org/3.9/library/const.html#True
770 https://docs.python.org/3.9/library/const.html#False
771 https://docs.python.org/3.9/library/functions.html#bool
772 https://docs.python.org/3.9/library/const.html#False
773 https://docs.python.org/3.9/library/const.html#None
774 https://docs.python.org/3.9/library/const.html#True
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

```python
class gpiozero.LEDCollection(*args, **kwargs)
```

Extends `CompositeOutputDevice` (page 186). Abstract base class for `LEDBoard` (page 157) and `LEDBarGraph` (page 160).

**Property is_lit**

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

**Property 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 187) with `on()` (page 186), `off()` (page 186), and `toggle()` (page 186) methods for controlling subordinate output devices. Also extends `value` (page 186) to be writeable.

**Parameters**

- `*args` (*Device* (page 199)) – The un-named devices that belong to the composite device. The `value` (page 199) attributes of these devices will be represented within the composite device’s tuple `value` (page 186) 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 186) 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 226) or `None`) – See `API - Pins` (page 221) for more information (this is an advanced feature which most users can ignore).

- `**kwargs` (*Device* (page 199)) – 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 186) attributes will be accessible as named elements of the composite device’s tuple `value` (page 186).

**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.

**Property 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.

---

775 https://docs.python.org/3.9/library/stdtypes.html#list
17.2.3 CompositeDevice

```python
class gpiozero.CompositeDevice(*args, **kwargs)
```

Extends Device (page 199). 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]
<gpiozero.LED object on pin GPIO2, active_high=True, is_active=False>
>>> d[1]
<gpiozero.LED object on pin GPIO3, active_high=True, is_active=False>
>>> d[2]
<gpiozero.LED object on pin GPIO4, active_high=True, is_active=False>
>>> d.btn
<gpiozero.Button object on pin GPIO17, pull_up=True, is_active=False>
>>> d.value
CompositeDeviceValue(device_0=False, device_1=False, device_2=False, btn=False)
```

Parameters

- **args** (Device (page 199)) – The un-named devices that belong to the composite device. The value (page 188) attributes of these devices will be represented within the composite device’s tuple value (page 188) 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 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 226) or None) – See API - Pins (page 221) for more information (this is an advanced feature which most users can ignore).

- ****kwargs** (Device (page 199)) – 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).

```
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()
```

(continues on next page)
Device (page 199) 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()
```

- **property closed**
  Returns `True` if the device is closed (see the `close()` (page 187) method). Once a device is closed you can no longer use any other methods or properties to control or query the device.

- **property is_active**
  Composite devices are considered “active” if any of their constituent devices have a “truthy” value.

- **property namedtuple**
  The `namedtuple()` type constructed to represent the value of the composite device. The `value` (page 188) attribute returns values of this type.

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

---

776 https://docs.python.org/3.9/library/stdtypes.html#list
777 https://docs.python.org/3.9/reference/compound_stmts.html#with
778 https://docs.python.org/3.9/library/constant.html#True
779 https://docs.python.org/3.9/library/collections.html#collections.namedtuple
780 https://docs.python.org/3.9/library/collections.html#collections.namedtuple
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 105) object is active when it’s pressed, and can be used to trigger other devices when its state changes, a TimeOfDay (page 190) object is active during a particular time period.

Consider the following code in which a Button (page 105) object is used to control an LED (page 125) 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 190) object is used to control an LED (page 125) 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 105) object, internal devices like the TimeOfDay (page 190) object has value (page 190), values, is_active (page 190), when_activated (page 190), and when_deactivated (page 191) 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(*args, **kwargs)

Extends PolledInternalDevice (page 196) 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 180) plug between 07:00AM and 08:00AM:

```python
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 to 10 seconds).
- **pin_factory** *(Factory or None)* – See API - Pins (page 221) for more information (this is an advanced feature which most users can ignore).

property end_time

The time of day after which the device will be considered inactive.

property is_active

Returns True if the device is currently active and False otherwise. This property is usually derived from value (page 190). Unlike value (page 190), this is always a boolean.

property start_time

The time of day after which the device will be considered active.

property utc

If True, use a naive UTC time reading for comparison instead of a local timezone reading.

property value

Returns 1 when the system clock reads between start_time (page 190) and end_time (page 190), and 0 otherwise. If start_time (page 190) is greater than end_time (page 190) (indicating a period that crosses midnight), then this returns 1 when the current time is greater than start_time (page 190) or less than end_time (page 190).
whenActivated
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.

whenDeactivated
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(*args, **kwargs)
Extends PolledInternalDevice (page 196) 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)
google.when_activated = led.on
google.when_deactivated = led.off

pause()
```

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 226) or None) – See API - Pins (page 221) for more information (this is an advanced feature which most users can ignore).

property host
The hostname or IP address to test whenever value (page 191) is queried.

property isActive
Returns True if the device is currently active and False otherwise. This property is usually derived from value (page 191). Unlike value (page 191), this is always a boolean.
property 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(*args, **kwargs)

Extends PolledInternalDevice (page 196) to provide a device which is active when the CPU temperature exceeds the threshold value.

The following example plots the CPU’s temperature on an LED bar graph:

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

# Use minimums and maximums that are closer to "normal" usage so the bar graph is a bit more "lively"
cpu = CPUTemperature(min_temp=50, max_temp=90)

print(f'Initial temperature: {cpu.temperature}C')

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 193) will read 0.0. This defaults to 0.0.
- **max_temp** (float) – The temperature at which value (page 193) will read 1.0. This defaults to 100.0.

---

791 https://docs.python.org/3.9/library/stdtypes.html#str
792 https://docs.python.org/3.9/library/constants.html#False
• **threshold** (*float*): The temperature above which the device will be considered “active”. (see *is_active* (page 193)). This defaults to 80.0.

• **event_delay** (*float*): The number of seconds between file reads (defaults to 5 seconds).

• **pin_factory** (*Factory* (page 226) or *None*): See *API - Pins* (page 221) for more information (this is an advanced feature which most users can ignore).

**property is_active**

Returns *True* when the CPU temperature (page 193) exceeds the threshold.

**property temperature**

Returns the current CPU temperature in degrees celsius.

**property 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 193) is *temperature* (page 193) 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.

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

**class gpiozero.LoadAverage(*args, **kwargs)**

Extends *PolledInternalDevice* (page 196) 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()
```

---

704 https://docs.python.org/3.9/library/stdtypes.html#str
705 https://docs.python.org/3.9/library/functions.html#float
706 https://docs.python.org/3.9/library/functions.html#float
707 https://docs.python.org/3.9/library/functions.html#float
708 https://docs.python.org/3.9/library/functions.html#float
709 https://docs.python.org/3.9/library/constants.html#True

---

18.1. Regular Classes 193
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 194) will read 0.0. This defaults to 0.0.

- **max_load_average** *(float)* – The load average at which value (page 194) 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 194)). 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 226) or None)* – See API - Pins (page 221) for more information (this is an advanced feature which most users can ignore).

**property is_active**

Returns True when the load_average (page 194) exceeds the threshold.

**property load_average**

Returns the current load average.

**property 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.

---

800 https://docs.python.org/3.9/library/stdtypes.html#str
801 https://docs.python.org/3.9/library/functions.html#float
802 https://docs.python.org/3.9/library/functions.html#float
803 https://docs.python.org/3.9/library/functions.html#float
804 https://docs.python.org/3.9/library/functions.html#int
805 https://docs.python.org/3.9/library/functions.html#float
806 https://docs.python.org/3.9/library/constants.html#True
18.1.5 DiskUsage

class gpiozero.DiskUsage(*args, **kwargs)

Extends PolledInternalDevice (page 196) 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(f'Current disk usage: {disk.usage}%')

graph = LEDBarGraph(5, 6, 13, 19, 25, pwm=True)
graph.source = disk
pause()
```

**Parameters**

- `filesystem (str)` – A path within the filesystem for which the disk usage needs to be computed. This defaults to `/`, which is the root filesystem.
- `threshold (float)` – The disk usage percentage above which the device will be considered “active” (see `is_active` (page 195)). This defaults to 90.0.
- `event_delay (float)` – The number of seconds between file reads (defaults to 30 seconds).
- `pin_factory (Factory or None)` – See API - Pins (page 221) for more information (this is an advanced feature which most users can ignore).

**property is_active**

Returns `True` when the disk usage (page 195) exceeds the threshold.

**property usage**

Returns the current disk usage in percentage.

**property value**

Returns the current disk usage as a value between 0.0 and 1.0 by dividing usage (page 195) 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):

```
Device → InternalDevice → PolledInternalDevice → CPU Temperature → Disk Usage → Load Average → Ping Server → Time Of Day
```

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(*args, **kwargs)
```

Extends `InternalDevice` (page 196) 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(*args, **kwargs)
```

Extends `Device` (page 199) 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.

References:
- `https://docs.python.org/3.9/library/stdtypes.html#str`
- `https://docs.python.org/3.9/library/functions.html#float`
- `https://docs.python.org/3.9/library/functions.html#float`
- `https://docs.python.org/3.9/library/consts.html#True`
The GPIO Zero class hierarchy is quite extensive. It contains several base classes (most of which are documented in their corresponding chapters):

- **Device** (page 199) is the root of the hierarchy, implementing base functionality like `close()` (page 199) and context manager handlers.
- **GPIODevice** (page 122) represents individual devices that attach to a single GPIO pin.
- **SPIDevice** (page 154) represents devices that communicate over an SPI interface (implemented as four GPIO pins).
- **InternalDevice** (page 196) represents devices that are entirely internal to the Pi (usually operating system related services).
- **CompositeDevice** (page 187) represents devices composed of multiple other devices like HATs.

There are also several **mixin classes**\(^{811}\) 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):

\(^{811}\) [https://en.wikipedia.org/wiki/Mixin](https://en.wikipedia.org/wiki/Mixin)
19.1 Device

class gpiozero.Device(*args, **kwargs)

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 199) property, the value (page 199) property, and the close() (page 199) 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 199) 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()
...    bz.off()

>>> with LED(16) as led:
...    led.on()
```

property closed

Returns True if the device is closed (see the close() (page 199) method). Once a device is closed you can no longer use any other methods or properties to control or query the device.

property is_active

Returns True if the device is currently active and False otherwise. This property is usually derived from value (page 199). Unlike value (page 199), this is always a boolean.

property 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

```python
class gpiozero.ValuesMixin(...)
```

Adds a `values` (page 200) property to the class which returns an infinite generator of readings from the `value` (page 199) 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.

**property values**

An infinite iterator of values read from `value`.

19.3 SourceMixin

```python
class gpiozero.SourceMixin(...)
```

Adds a `source` (page 200) property to the class which, given an iterable or a `ValuesMixin` (page 200) descendent, sets `value` (page 199) 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.

**property source**

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

**property source_delay**

The delay (measured in seconds) in the loop used to read values from `source` (page 200). Defaults to 0.01 seconds which is generally sufficient to keep CPU usage to a minimum while providing adequate responsiveness.

19.4 SharedMixin

```python
class gpiozero.SharedMixin(...)
```

This mixin marks a class as “shared”. In this case, the meta-class (GPIOMeta) will use `_shared_key()` (page 200) 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 199) is called, an internal reference counter will be decremented and the instance will only close when it reaches zero.

```python
classmethod _shared_key(*args, **kwargs)
```

This is called with the constructor arguments to generate a unique key (which must be storable in a `dict`\(^\text{816}\) and, thus, immutable and hashable) representing the instance that can be shared. This must be overridden by descendents.

---

\(^\text{812}\) https://docs.python.org/3.9/reference/compound_stmts.html#with
\(^\text{813}\) https://docs.python.org/3.9/library/constants.html#True
\(^\text{814}\) https://docs.python.org/3.9/library/constants.html#True
\(^\text{815}\) https://docs.python.org/3.9/library/constants.html#False
\(^\text{816}\) https://docs.python.org/3.9/library/stdtypes.html#dict
19.5 EventsMixin

class gpiozero.EventsMixin(...)

Adds edge-detected when_activated() (page 201) and when_deactivated() (page 201) events to a device based on changes to the is_active (page 199) property common to all devices. Also adds wait_for_active() (page 201) and wait_for_inactive() (page 201) methods for level-waiting.

Note: Note that this mixin provides no means of actually firing its events; call _fire_events() in subclasses when device state changes to trigger the events. This should also be called once at the end of initialization to set initial states.

wait_for_active (timeout=None)

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

Parameters

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

wait_for_inactive (timeout=None)

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

Parameters

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

property active_time

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

property inactive_time

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

when_activated

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

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

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

817 https://docs.python.org/3.9/library/functions.html#float
818 https://docs.python.org/3.9/library/constants.html#None
819 https://docs.python.org/3.9/library/functions.html#float
820 https://docs.python.org/3.9/library/constants.html#None
821 https://docs.python.org/3.9/library/constants.html#None
822 https://docs.python.org/3.9/library/constants.html#None
823 https://docs.python.org/3.9/library/constants.html#None
824 https://docs.python.org/3.9/library/constants.html#None
19.6 HoldMixin

class gpiozero.HoldMixin(...)

- Extends `EventsMixin` (page 201) to add the `when_held` (page 202) event and the machinery to fire that event repeatedly (when `hold_repeat` (page 202) is `True`) at internals defined by `hold_time` (page 202).

- **property 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 202) event rather than when the device activated, in contrast to `active_time` (page 201). If the device is not currently held, this is `None`.

- **property hold_repeat**
  - If `True` (page 202), `when_held` (page 202) will be executed repeatedly with `hold_time` (page 202) seconds between each invocation.

- **property hold_time**
  - The length of time (in seconds) to wait after the device is activated, until executing the `when_held` (page 202) handler. If `hold_repeat` (page 202) is True, this is also the length of time between invocations of `when_held` (page 202).

- **property is_held**
  - When `True`, the device has been active for at least `hold_time` (page 202) seconds.

- **when_held**
  - The function to run when the device has remained active for `hold_time` (page 202) 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.

---

825 https://docs.python.org/3.9/library/constants.html#True
826 https://docs.python.org/3.9/library/constants.html#None
827 https://docs.python.org/3.9/library/constants.html#True
828 https://docs.python.org/3.9/library/constants.html#True
829 https://docs.python.org/3.9/library/constants.html#None
Chapter Twenty

API - Device Source Tools

GPIO Zero includes several utility routines which are intended to be used with the Source/Values (page 61) 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 200) and values (page 200) deal with infinite iterators, another excellent source of utilities is the itertools\(^\text{830}\) module in the standard library.

20.1 Single source conversions

**gpiozero.tools.absolutely(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 absolutely, scaled
from signal import pause

led = PWMLED(4)
motor = Motor(22, 27)
pot = MCP3008(channel=0)

motor.source = scaled(pot, -1, 1)
led.source = absolutely(motor)

pause()
```

**gpiozero.tools.booleanized(values, min_value, max_value, hysteresis=0)**

Returns True for each item in values between min_value and max_value, and False otherwise. hysteresis can optionally be used to add hysteresis\(^\text{831}\) 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:

```python
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()
```

\(^\text{830}\) https://docs.python.org/3.9/library/itertools.html#module-itertools

\(^\text{831}\)
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)
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 205), repeat_after can’t be 0. For example, to block every tenth item read from an ADC:

---

831 https://en.wikipedia.org/wiki/Hysteresis
832 https://docs.python.org/3.9/library/constants.html#True
833 https://docs.python.org/3.9/library/constants.html#False
834 https://docs.python.org/3.9/library/constants.html#False
835 https://docs.python.org/3.9/library/constants.html#True
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:

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:

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):

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:

from gpiozero import LEDBoard, Button
from gpiozero.tools import queued
from signal import pause

leds = LEDBoard(5, 6, 13, 19, 26)
btn = Button(17)

(continues on next page)
for i in range(4):
    leds[i].source = queued(leds[i + 1], 5)
    leds[i].source_delay = 0.01

leds[4].source = btn
pause()

`gpiozero.tools.smoothed(values, qsize, average=<function mean>)`

Queues up readings from `values` (the number of readings queued is determined by `qsize`) and begins yielding the average of the last `qsize` values when the queue is full. The larger the `qsize`, the more the values are smoothed. For example, to smooth the analog values read from an ADC:

```python
from gpiozero import MCP3008
from gpiozero.tools import smoothed

adc = MCP3008(channel=0)

for value in smoothed(adc, 5):
    print(value)
```

`gpiozero.tools.scaled(values, output_min, output_max, input_min=0, input_max=1)`

Returns values scaled from `output_min` to `output_max`, assuming that all items in values lie between `input_min` and `input_max` (which default to 0 and 1 respectively). For example, to control the direction of a motor (which is represented as a value between -1 and 1) using a potentiometer (which typically provides values between 0 and 1):

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

motor = Motor(20, 21)
pot = MCP3008(channel=0)

motor.source = scaled(pot, -1, 1)
pause()
```

**Warning:** If `values` contains elements that lie outside `input_min` to `input_max` (inclusive) then the function will not produce values that lie within `output_min` to `output_max` (inclusive).

## 20.2 Combining sources

`gpiozero.tools.all_values(*values)`

Returns the logical conjunction of all supplied values (the result is only True if and only if all input values are simultaneously True). One or more `values` can be specified. For example, to light an LED (page 125) 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)
```

(continues on next page)
led.source = all_values(btn1, btn2)
pause()

**gpiozero.tools.any_values**(*values*)

Returns the logical disjunction\(^{39}\) of all supplied values (the result is True\(^{40}\) if any of the input values are currently True\(^{41}\)). One or more values can be specified. For example, to light an LED (page 125) 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 127) as the average of several potentiometers connected to an MCP3008 (page 149) ADC:

```python
from gpiozero import MCP3008, PWMLED
from gpiozero.tools import averaged
from signal import pause

pot1 = MCP3008(channel=0)
pot2 = MCP3008(channel=1)
pot3 = MCP3008(channel=2)

led = PWMLED(4)

led.source = averaged(pot1, pot2, pot3)
pause()
```

**gpiozero.tools.multiplied**(*values*)

Returns the product of all supplied values. One or more values can be specified. For example, to light a PWMLED (page 127) as the product (i.e. multiplication) of several potentiometers connected to an MCP3008 (page 149) 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)
```

---

\(^{39}\) [https://en.wikipedia.org/wiki/Logical_disjunction](https://en.wikipedia.org/wiki/Logical_disjunction)

\(^{40}\) [https://docs.python.org/3.9/library/constants.html#True](https://docs.python.org/3.9/library/constants.html#True)

\(^{41}\) [https://docs.python.org/3.9/library/constants.html#True](https://docs.python.org/3.9/library/constants.html#True)
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 127) as the (scaled) sum of several potentiometers connected to an MCP3008 (page 149) ADC:

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

pot1 = MCP3008(channel=0)
pot2 = MCP3008(channel=1)
pot3 = MCP3008(channel=2)
led = PWMLED(4)

led.source = scaled(summed(pot1, pot2, pot3), 0, 1, 0, 3)
pause()
```

gpiozero.tools.zip_values(*devices)

Provides a source constructed from the values of each item, for example:

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

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

robot.source = zip_values(left, right)
pause()
```

zip_values(left, right) is equivalent to zip(left.values, right.values).

## 20.3 Artificial sources

gpiozero.tools.alternating_values(*initial_value=False)

Provides an infinite source of values alternating between True\(^\text{842}\) and False\(^\text{843}\), starting with initial_value (which defaults to False\(^\text{844}\)). 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()
```

\(^{842}\) https://docs.python.org/3.9/library/constants.html#True
\(^{843}\) https://docs.python.org/3.9/library/constants.html#False
\(^{844}\) https://docs.python.org/3.9/library/constants.html#False
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 \textit{scaled()} (page 206).

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 \textit{scaled()} (page 206).

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 \textit{scaled()} (page 206).

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)
```

(continues on next page)
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 206).
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 162) or `LEDMultiCharDisplay` (page 163)).

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[^7], returning a dict[^8] suitable for use with `LEDCharDisplay` (page 162).

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_

... |.| ..| _._ |._ _| |_._ ..| ..| ..| ..| ..| _._
... |_| ..| |._ ._| ..| ._| |_| ..| |_| ._|
```

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.

`gpiozero.fonts.load_font_14seg(filename_or_obj)`

Given a filename or a file-like object, parse it as a font definition for a 14-segment display[^9], returning a dict[^10] suitable for use with `LEDCharDisplay` (page 162).

[^7]: https://en.wikipedia.org/wiki/Seven-segment_display
[^8]: https://docs.python.org/3.9/library/stdtypes.html#dict
[^9]: DOI: 10.1002/9781119135345.ch21
[^10]: DOI: 10.1002/9781119135345.ch21
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 5x5 characters laid out as follows:

```
X.a..
fijkb
.g.h.
elmnc
..d..
```

Where X is the character being defined, and a-n define the states of the LEDs for that position. a, d, g, and h are on if they are “-”, b, c, e, f, j, and m are on if they are “|”. i and n are on if they are “”. Finally, k and l 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---
.... | /| /| /| | | | | | | |
.... | | | --- -- ---| --- |--- | --- ---|
.... |/ | | | | | | | | | | |
.... --- --- --- --- --- --- ---
```

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-segment or 14-segment displays you don’t want this function; see `load_font_7seg()` (page 211) or `load_font_14seg()` (page 211) 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”.

For example, for 7-segment displays this function is called as follows:

```
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:

```
012
345  => '012345678'
678
```
Position 0 is always assumed to be the character being defined. The \textit{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.

\begin{footnotesize}
\begin{itemize}
\item \textsuperscript{849} https://en.wikipedia.org/wiki/Seven-segment_display
\item \textsuperscript{850} https://en.wikipedia.org/wiki/Fourteen-segment_display
\end{itemize}
\end{footnotesize}
GPIO Zero includes a `Tone` (page 215) class intended for use with the `TonalBuzzer` (page 133). 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.

The `Tone` (page 215) class can be used with the `TonalBuzzer` (page 133) class to easily represent musical tones. The class can be constructed in a variety of ways. For example as a straight frequency in Hz (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 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 216), `from_midi()` (page 216), or `from_note()` (page 216), or you can specify a keyword argument when constructing:

```python
>>> Tone.from_frequency(440)
>>> Tone.from_midi(69)
>>> Tone.from_note('A4')
```

Several attributes are provided to permit conversion to any of the supported construction formats: `frequency` (page 216), `midi` (page 216), and `note` (page 216). Methods are provided to step `up()` (page 216) or `down()` (page 215) to adjacent MIDI notes.

**Warning:** Currently `Tone` (page 215) 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.
down \((n=1)\)
Return the \textit{Tone} (page 215) \textit{n} semi-tones below this frequency \((n \text{ defaults to 1})\).

\textbf{classmethod from\_frequency} \((freq)\)
Construct a \textit{Tone} (page 215) from a frequency specified in Hz\(^{854}\) which must be a positive floating-point value in the range \(0 < \text{freq} \leq 20000\).

\textbf{classmethod from\_midi} \((midi\_note)\)
Construct a \textit{Tone} (page 215) from a MIDI note, which must be an integer in the range 0 to 127. For reference, A4 (concert A\(^{855}\) typically used for tuning) is MIDI note #69.

\textbf{classmethod from\_note} \((note)\)
Construct a \textit{Tone} (page 215) 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\(^{856}\), the typical tuning note at 440Hz, would be represented as “A4”. One semi-tone above this would be “A#4” or alternatively “Bb4”. Unicode representations of sharp and flat are also accepted.

\textbf{up} \((n=1)\)
Return the \textit{Tone} (page 215) \textit{n} semi-tones above this frequency \((n \text{ defaults to 1})\).

\textbf{property frequency}
Return the frequency of the tone in Hz\(^{857}\).

\textbf{property midi}
Return the (nearest) MIDI note to the tone’s frequency. This will be an integer number in the range 0 to 127. If the frequency is outside the range represented by MIDI notes (which is approximately 8Hz to 12.5KHz) \texttt{ValueError}\(^{858}\) exception will be raised.

\textbf{property note}
Return the (nearest) note to the tone’s frequency. This will be a string in the form accepted by \texttt{from\_note()} (page 216). If the frequency is outside the range represented by this format (“A0” is approximately 27.5Hz, and “G9” is approximately 12.5Khz) a \texttt{ValueError}\(^{859}\) exception will be raised.

---

\(^{851}\) https://en.wikipedia.org/wiki/Hertz
\(^{852}\) https://en.wikipedia.org/wiki/Concert_pitch
\(^{853}\) https://docs.python.org/3.9/library/functions.html#float
\(^{854}\) https://en.wikipedia.org/wiki/Hertz
\(^{855}\) https://en.wikipedia.org/wiki/Concert_pitch
\(^{856}\) https://en.wikipedia.org/wiki/Concert_pitch
\(^{857}\) https://en.wikipedia.org/wiki/Hertz
\(^{858}\) https://docs.python.org/3.9/library/exceptions.html#ValueError
\(^{859}\) https://docs.python.org/3.9/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

`gpiozero.pi_info(revision=None)`

Deprecated function for retrieving information about a `revision` of the Raspberry Pi. If you wish to retrieve information about the board that your script is running on, please query the `Factory.board_info` (page 227) property like so:

```python
>>> from gpiozero import Device
>>> Device.ensure_pin_factory()
>>> Device.pin_factory.board_info
PiBoardInfo(revision='a02082', model='3B', pcb_revision='1.2',
released='2016Q1', soc='BCM2837', manufacturer='Sony', memory=1024,
storage='MicroSD', usb=4, usb3=0, ethernet=1, eth_speed=100, wifi=True,
bluetooth=True, csi=1, dsi=1, headers=..., board=...)
```

To obtain information for a specific Raspberry Pi board revision, use the `PiBoardInfo.from_revision()` constructor.

**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 (revision, model, pcb_revision, released, soc, manufacturer, memory,
storage, usb, usb3, ethernet, eth_speed, wifi, bluetooth, csi, dsi, headers,
board)
```

---

860 https://docs.python.org/3.9/library/stdtypes.html#str
861 https://docs.python.org/3.9/library/constants.html#None
23.3 HeaderInfo

```python
class gpiozero.HeaderInfo(name, rows, columns, pins)
```

This class is a named tuple 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.pins.native import NativeFactory

factory = NativeFactory()
j8 = factory.board_info.headers['J8']
print(f'{j8:full}')
p1 = factory.board_info.headers['P1']
print(f'{p1:col2}')
print(f'{p1:row1}')
```

“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. “rev” can be added to output the row or column in reverse order:

```python
# force use of ANSI codes
j8 = factory.board_info.headers['J8']
print(f'{j8:color row2}')
# force plain ASCII
print(f'{j8:mono row2}')
# output in reverse order
print(f'{j8:color rev row1}')
```

The following attributes are defined:

- **pprint(color=None)**
  - Pretty-print a diagram of the header pins.
  - If `color` is `None` (the default, the diagram will include ANSI color codes if stdout is a color-capable terminal). Otherwise `color` can be set to `True` or `False` to force color or monochrome output.

- **name**
  - The name of the header, typically as it appears silk-screened on the board (e.g. “P1” or “J8”).

- **rows**
  - The number of rows on the header.

- **columns**
  - The number of columns on the header.

- **pins**
  - A dictionary mapping physical pin numbers to `PinInfo` (page 219) tuples.
23.4 PinInfo

class gpiozero.PinInfo (number, name, names, pull, row, col, interfaces)

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).

- **name**
  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 board’s numbering scheme), etc.

- **names**
  A set of all the names that can be used to identify this pin with BoardInfo.find_pin(). The name (page 219) attribute is the “typical” name for this pin, and will be one of the values in this set.

  When “gpio” is in interfaces (page 219), these names can be used with Factory.pin() (page 226) to construct a Pin (page 227) instance representing this pin.

- **pull**
  A string indicating the fixed pull of the pin, if any. This is a blank string if the pin has no fixed pull, but may be “up” in the case of pins typically used for I2C such as GPIO2 and GPIO3 on a Raspberry Pi.

- **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)

- **interfaces**
  A dict mapping interfaces that this pin can be a part of to the description of that pin in that interface (e.g. “i2c” might map to “I2C0 SDA”). Typical keys are “gpio”, “spi”, “i2c”, “uart”, “pwm”, “smi”, and “dpi”.

- **pull_up**
  Deprecated variant of pull (page 219).

- **function**
  Deprecated alias of name (page 219).

---

867 https://docs.python.org/3.9/library/collections.html#collections.namedtuple
868 https://docs.python.org/3.9/library/stdtypes.html#dict
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 226) which turns it into a Pin (page 227) implementation. The default factory can be queried (and changed) with Device.pin_factory (page 199). 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 45) chapter).

This is illustrated in the following flow-chart:
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 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
>>> led = LED(2)
```

(continues on next page)

---

869 https://docs.python.org/3.9/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 `RPiGPIOFactory` (page 235).

On a PC (with no pin libraries installed and no environment variables set), importing will work but attempting to create a device will raise `BadPinFactory` (page 240):

```
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 `GPIOZERO_PIN_FACTORY` (page 77) environment variable. For example:

```
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 `GPIOZERO_PIN_FACTORY` (page 77) for the rest of your session you can `export` this value:

```
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()
```

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 226) and Pin (page 227) classes are listed in the table below. Factories are listed in the order that they are tried by default.

<table>
<thead>
<tr>
<th>Name</th>
<th>Factory class</th>
<th>Pin class</th>
</tr>
</thead>
<tbody>
<tr>
<td>rpigpio</td>
<td><code>gpiozero.pins.rpigpio.RPiGPIOFactory</code></td>
<td><code>gpiozero.pins.rpigpio.RPiGPIOPin</code></td>
</tr>
<tr>
<td>lgpio</td>
<td><code>gpiozero.pins.lgpio.LGPIOFactory</code></td>
<td><code>gpiozero.pins.lgpio.LGPIOPin</code></td>
</tr>
<tr>
<td>pigpio</td>
<td><code>gpiozero.pins.pigpio.PiGPIOFactory</code></td>
<td><code>gpiozero.pins.pigpio.PiGPIOPin</code></td>
</tr>
<tr>
<td>native</td>
<td><code>gpiozero.pins.native.NativeFactory</code></td>
<td><code>gpiozero.pins.native.NativePin</code></td>
</tr>
</tbody>
</table>

If you need to change the default pin factory from within a script, either set `Device.pin_factory` (page 199) 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.PiGPIOPin` (page 236) on a remote pi called "remote-pi" you can set the `PIGPIO_ADDR` (page 77) environment variable when running your script:

```bash
$ GPIOZERO_PIN_FACTORY=pigpio PIGPIO_ADDR=remote-pi python3 my_script.py
```

Like the `GPIOZERO_PIN_FACTORY` (page 77) 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 235) for one pin, and `NativeFactory` (page 237) 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 45).
24.2 Mock pins

There’s also a MockFactory (page 237) 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:

```python
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:

```python
>>> 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 238) to be able to use devices requiring PWM:
or:

```python
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\(^{870}\) 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. Descendants *must* override the following methods:

- `ticks()` (page 227)
- `ticks_diff()` (page 227)
- `_get_board_info()`

Descendants *may* override the following methods, if applicable:

- `close()` (page 226)
- `reserve_pins()` (page 226)
- `release_pins()` (page 226)
- `release_all()` (page 226)
- `pin()` (page 226)
- `spi()` (page 227)

**`close()`**

Closes the pin factory. This is expected to clean up all resources manipulated by the factory. It is typically called at script termination.

**`pin(name)`**

Creates an instance of a *Pin* (page 227) descendent representing the specified pin.

**Warning:** Descendants must ensure that pin instances representing the same hardware are identical; i.e. two separate invocations of `pin()` (page 226) 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 226) for further information.

**`release_pins(reserver, *names)`**

Releases the reservation of `reserver` against pin `names`. This is typically called during `close()` (page 199) 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).

\(^{870}\) [https://github.com/gpiozero/gpiozero/tree/master/tests](https://github.com/gpiozero/gpiozero/tree/master/tests)
**reserve_pins** *(requester, *names)*

Called to indicate that the device reserves the right to use the specified pin names. 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 226)*.

**spi** *(**spi_args)**

Returns an instance of an SPI *(page 230)* 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 240)*.

**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 227)* method.

**ticks_diff** *(later, earlier)*

Return the time in seconds between two ticks *(page 227)* 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 227)*.

**property board_info**

Returns a BoardInfo instance (or derivative) representing the board that instances generated by this factory will be attached to.

**class gpiozero.Pin**

Abstract base class representing a pin attached to some form of controller, be it GPIO, SPI, ADC, etc.

Descendents should override property getters and setters to accurately represent the capabilities of pins. Descendents **must** override the following methods:

- **_get_info** *
- **_get_function** *
- **_set_function** *
- **_get_state** *

Descendents **may** additionally override the following methods, if applicable:

- **close** *(page 227)*
- **output_with_state** *(page 228)*
- **input_with_pull** *(page 228)*
- **_set_state** *
- **_get_frequency** *
- **_set_frequency** *
- **_get_pull** *
- **_set_pull** *
- **_get_bounce** *
- **_set_bounce** *
- **_get_edges** *
- **_set_edges** *
- **_get_when_changed** *
- **_set_when_changed** *
close()

Cleans up the resources allocated to the pin. After this method is called, this Pin (page 227) 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).

property 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 228) is currently “rising”, bounce (page 228) is currently 5/1000 (5ms), then the waveform below will only fire when_changed (page 229) on two occasions despite there being three rising edges:

If the pin does not support edge detection, attempts to set this property will raise PinEdgeDetectUn-supported (page 242). If the pin supports edge detection, the class must implement bounce detection, even if only in software.

property edges

The edge that will trigger execution of the function or bound method assigned to when_changed (page 229). This can be one of the strings “both” (the default), “rising”, “falling”, or “none”:

(continues on next page)
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 `PinEdgeDetectionUnsupported` (page 242).

**property 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 243) will be raised when attempting to set this to a value other than `None`.

**property 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 242) will be raised.

**property info**

Returns the `PinInfo` (page 219) associated with the pin. This can be used to determine physical properties of the pin, including its location on the header, fixed pulls, and the various specs that can be used to identify it.

**property 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 242). If the specified value is not supported by the underlying hardware, `PinInvalidPull` (page 242) is raised.

**property 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 229) 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 229) is not `None`) return a value between 0.0 and 1.0 representing the current PWM duty cycle.

If a pin is currently configured for input, and an attempt is made to set this attribute, `PinSetInput` (page 242) will be raised. If an invalid value is specified for this attribute, `PinInvalidState` (page 242) will be raised.

**property when_changed**

A function or bound method to be called when the pin’s state changes (more specifically when the edge specified by `edges` (page 228) is detected on the pin). The function or bound method must accept two parameters: the first will report the ticks (from `Factory.ticks()` (page 227)) when the pin’s state changed, and the second will report the pin’s current state.

24.3. Base classes
Warning: Depending on hardware support, the state is not guaranteed to be accurate. For instance, many GPIO implementations will provide an interrupt indicating when a pin’s state changed but not what it changed to. In this case the pin driver simply reads the pin’s current state to supply this parameter, but the pin’s state may have changed since the interrupt. Exercise appropriate caution when relying upon this parameter.

If the pin does not support edge detection, attempts to set this property will raise `PinEdgeDetectionUnsupported` (page 242).

```python
class gpiozero.SPI(*args, **kwargs)
```

Abstract interface for Serial Peripheral Interface (SPI) implementations. Descendents must override the following methods:

- `transfer()` (page 230)
- `get_clock_mode()`

Descendents may override the following methods, if applicable:

- `read()` (page 230)
- `write()` (page 230)
- `set_clock_mode()`
- `get_lsb_first()`
- `set_lsb_first()`
- `get_select_high()`
- `set_select_high()`
- `get_bits_per_word()`
- `set_bits_per_word()`

```python
read(n)
```

Read `n` words of data from the SPI interface, returning them as a sequence of unsigned ints, each no larger than the configured `bits_per_word` (page 230) of the interface.

This method is typically used with read-only devices that feature half-duplex communication. See `transfer()` (page 230) for full duplex communication.

```python
transfer(data)
```

Write `data` to the SPI interface. `data` must be a sequence of unsigned integer words each of which will fit within the configured `bits_per_word` (page 230) of the interface. The method returns the sequence of words read from the interface while writing occurred (full duplex communication).

The length of the sequence returned dictates the number of words of `data` written to the interface. Each word in the returned sequence will be an unsigned integer no larger than the configured `bits_per_word` (page 230) of the interface.

```python
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 230) 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 230) for full duplex communication.

---

871 https://docs.python.org/3.9/library/constants.html#None
872 https://docs.python.org/3.9/library/constants.html#None
873 https://docs.python.org/3.9/library/constants.html#None
874 https://docs.python.org/3.9/library/constants.html#None
875 https://docs.python.org/3.9/library/constants.html#None
**property 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.

**property clock_mode**

Presents a value representing the `clock_polarity` (page 231) and `clock_phase` (page 231) 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 231) and `clock_phase` (page 231) attributes simultaneously.

**property clock_phase**

The phase of the SPI clock pin. If this is `False` (the default), data will be read from the MISO pin when the clock pin activates. Setting this to `True` will cause data to be read from the MISO pin when the clock pin deactivates. On many data sheets this is documented as the CPHA value. Whether the clock edge is rising or falling when the clock is considered activated is controlled by the `clock_polarity` (page 231) attribute (corresponding to CPOL).

The following diagram indicates when data is read when `clock_polarity` (page 231) is `False`, and `clock_phase` (page 231) is `False` (the default), equivalent to CPHA 0:

```
CLK | | | | | | | | | | | | | |
| | | | | | | | | | | | | |
----' `---' `---' `---' `---' `---' `---' `-------
MISO---. ,---. ,---. ,---. ,---. ,---. ,---. \
-----{ Bit X Bit X Bit X Bit X Bit X Bit X Bit }------
```

The following diagram indicates when data is read when `clock_polarity` (page 231) is `False`, but `clock_phase` (page 231) is `True`, equivalent to CPHA 1:

```
CLK | | | | | | | | | | | | | |
| | | | | | | | | | | | | |
----' `---' `---' `---' `---' `---' `---' `-------
MISO---. ,---. ,---. ,---. ,---. ,---. ,---. \
-----{ Bit X Bit X Bit X Bit X Bit X Bit X Bit }------
```

**property 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 231) is `False` (the default), equivalent to CPOL 0:
The following diagram illustrates the waveform when \texttt{clock_polarity} (page 231) is \texttt{True}, equivalent to CPOL 1:

```
  idle off off off off off off idle
------' `---' `---' `---' `---' `---' `---'
  CLK | | | | | | | |
```

\textbf{property lsb\textunderscore first}

Controls whether words are read and written LSB in (Least Significant Bit first) order. The default is \texttt{False}, indicating that words are read and written in MSB (Most Significant Bit first) order. Effectively, this controls the Bitendianness of the connection.

The following diagram shows the a word containing the number 5 (binary 0101) transmitted on MISO with \texttt{bits\_per\_word} (page 230) set to 4, and \texttt{clock\_mode} (page 231) set to 0, when \texttt{lsb\_first} (page 232) is \texttt{False} (the default):

```
   MSB : : MSB
----------' : `-------' : `----
MISO: | : | : | : |
     : | : | : | : |
     : | : | : | : |
     : | : | : | : |
     : | : | : | : |
     : | : | : | : |
MISO: | | | | | |
```

And now with \texttt{lsb\_first} (page 232) set to \texttt{True} (and all other parameters the same):

```
   MSB : : MSB
----------' : `-------' : `----
MISO: | : | : | : |
     : | : | : | : |
     : | : | : | : |
     : | : | : | : |
     : | : | : | : |
     : | : | : | : |
MISO: | | | | | |
```

\textbf{property rate}

Controls the speed of the SPI interface in Hz (or baud).

Note that most software SPI implementations ignore this property, and will raise \texttt{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).

\textbf{property select\_high}

If \texttt{False} (the default), the chip select line is considered active when it is pulled low. When set to \texttt{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 231) is False\(^93\), and `select_high` (page 232) is False\(^94\) (the default):

And when `select_high` (page 232) is True\(^95\):

```plaintext
class gpiozero.pins.pi.PiFactory

Extends Factory (page 226). Abstract base class representing hardware attached to a Raspberry Pi. This forms the base of LocalPiFactory (page 234).

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(name)

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 233) for the same pin specification must return the same object.

---


\(^{94}\) [https://docs.python.org/3.9/library/constants.html#False](https://docs.python.org/3.9/library/constants.html#False)

\(^{95}\) [https://docs.python.org/3.9/library/constants.html#True](https://docs.python.org/3.9/library/constants.html#True)
**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 240).

If the pins specified match the hardware SPI pins (clock on GPIO11, MOSI on GPIO10, MISO on GPIO9, and chip select on GPIO8 or GPIO7), and the spidev module can be imported, a hardware based interface (using spidev) will be returned. Otherwise, a software based interface will be returned which will use simple bit-banging to communicate.

Both interfaces have the same API, support clock polarity and phase attributes, and can handle half and full duplex communications, but the hardware interface is significantly faster (though for many simpler devices this doesn’t matter).

**class gpiozero.pins.pi.PiPin**(/**factory**, /**info**)

Extends Pin (page 227). 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()

**property info**

Returns the PinInfo associated with the pin. This can be used to determine physical properties of the pin, including its location on the header, fixed pulls, and the various specs that can be used to identify it.

**class gpiozero.pins.local.LocalPiFactory**

Extends PiFactory (page 233). Abstract base class representing pins attached locally to a Pi. This forms the base class for local-only pin interfaces (RPiGPIOPin (page 235), LGPIOPin (page 236), and NativePin (page 237)).

**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 235) method.

```python
static ticks_diff (later, earlier)
```

Return the time in seconds between two `ticks()` (page 234) 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 234).

```python
class gpiozero.pins.local.LocalPiPin (factory, info)
```

Extends `PiPin` (page 234). Abstract base class representing a multi-function GPIO pin attached to the local Raspberry Pi.

### 24.4 RPi.GPIO

```python
class gpiozero.pins.rpigpio.RPiGPIOFactory
```

Extends `LocalPiFactory` (page 234). 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:

```python
from gpiozero.pins.rpigpio import RPiGPIOFactory
from gpiozero import LED

factory = RPiGPIOFactory()
led = LED(12, pin_factory=factory)
```

```python
class gpiozero.pins.rpigpio.RPiGPIOPin (factory, info)
```

Extends `LocalPiPin` (page 235). Pin implementation for the RPi.GPIO library. See `RPiGPIOFactory` (page 235) for more information.

### 24.5 lgpio

```python
class gpiozero.pins.lgpio.LGPIOFactory (chip=None)
```

Extends `LocalPiFactory` (page 234). Uses the `lgpio` 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:

```python
from gpiozero.pins.lgpio import LGPIOFactory
from gpiozero import LED

factory = LGPIOFactory(chip=0)
led = LED(12, pin_factory=factory)
```

---

896 [https://pypi.python.org/pypi/RPi.GPIO](https://pypi.python.org/pypi/RPi.GPIO)
897 [https://pypi.python.org/pypi/RPi.GPIO](https://pypi.python.org/pypi/RPi.GPIO)
898 [https://pypi.python.org/pypi/lgio](https://pypi.python.org/pypi/lgio)
The *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 /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).

```python
class gpiozero.pins.lgpio.LGPIOPin(factory, info)
Extends LocalPiPin (page 235). Pin implementation for the lgpio library. See LGPIOFactory
(page 235) for more information.
```

### 24.6 PiGPIO

```python
class gpiozero.pins.pigpio.PiGPIOFactory (host=None, port=None)
Extends PiFactory (page 233). Uses the pigpio library to interface to the Pi’s GPIO pins. The pigpio library relies on a daemon (pigpiod) to be running as root to provide access to the GPIO pins, and communicates with this daemon over a network socket.

While this does mean only the daemon itself should control the pins, the architecture does have several advantages:

- Pins can be remote controlled from another machine (the other machine doesn’t even have to be a Raspberry Pi; it simply needs the pigpio client library installed on it)
- The daemon supports hardware PWM via the DMA controller
- Your script itself doesn’t require root privileges; it just needs to be able to communicate with the daemon

You can construct 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, info)
Extends PiPin (page 234). Pin implementation for the pigpio library. See PiGPIOFactory (page 236)
for more information.
```

---

898 [http://abyz.me.uk/lg/py_lgpio.html](http://abyz.me.uk/lg/py_lgpio.html)
899 [http://abyz.me.uk/lg/py_lgpio.html](http://abyz.me.uk/lg/py_lgpio.html)
900 [http://abyz.me.uk/rpi/pigpio/](http://abyz.me.uk/rpi/pigpio/)
901 [http://abyz.me.uk/rpi/pigpio/](http://abyz.me.uk/rpi/pigpio/)
902 [http://abyz.me.uk/rpi/pigpio/](http://abyz.me.uk/rpi/pigpio/)
24.7  Native

```python
class gpiozero.pins.native.NativeFactory

Extends LocalPiFactory (page 234). 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, info)

Extends LocalPiPin (page 235). Native pin implementation. See NativeFactory (page 237) for more information.
```

```python
class gpiozero.pins.native.Native2835Pin(factory, info)

Extends NativePin (page 237) for Pi hardware prior to the Pi 4 (Pi 0, 1, 2, 3, and 3+).
```

```python
class gpiozero.pins.native.Native2711Pin(factory, info)

Extends NativePin (page 237) for Pi 4 hardware (Pi 4, CM4, Pi 400 at the time of writing).
```

24.8  Mock

```python
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.board_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 237) method by default. This can be changed after construction by modifying the `pin_class` (page 237) attribute.

```python
pin_class

This attribute stores the MockPin (page 238) class (or descendant) that will be used when constructing pins with the `pin()` (page 237) method (if no `pin_class` parameter is used to override it). It defaults on construction to the value of the `pin_class` parameter in the constructor, or MockPin (page 238) if that is unspecified.
```

```python
pin(name, pin_class=None, **kwargs)

The pin method for MockFactory (page 237) additionally takes a `pin_class` attribute which can be used to override the class' `pin_class` (page 237) attribute. Any additional keyword arguments will be passed along to the pin constructor (useful with things like MockConnectedPin (page 238) which expect to be constructed with another pin).
```

```python
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 238) method.

static ticks_diff(later, earlier)

Return the time in seconds between two ticks() (page 237) 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 237).

class gpiozero.pins.mock.MockPin(factory, info)

A mock pin used primarily for testing. This class does not support PWM.

class gpiozero.pins.mock.MockPWMPin(factory, info)

This derivative of MockPin (page 238) adds PWM support.

class gpiozero.pins.mock.MockConnectedPin(factory, info, input_pin=None)

This derivative of MockPin (page 238) 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, info, charge_time=0.01)

This derivative of MockPin (page 238) 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, info, echo_pin=None, echo_time=0.04)

This derivative of MockPin (page 238) is intended to be used with another MockPin (page 238) 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.

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 descendants can then be passed as the spi_class parameter of the MockFactory (page 237) constructor to have instances attached to any SPI interface requested by an SPIDevice.
API - EXCEPTIONS

The following exceptions are defined by GPIO Zero. Please note that multiple inheritance is heavily used in the exception hierarchy to make testing for exceptions easier. For example, to capture any exception generated by GPIO Zero’s code:

```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 239), 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 145) you would expect a `ValueError` to be raised. In fact, a `OutputDeviceBadValue` error will be raised. However, note that this descends from both `GPIOZeroError` (page 239) (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

- **exception** `gpiozero.DeviceClosed`
  
  Bases: `GPIOZeroError` (page 239)
  
  Error raised when an operation is attempted on a closed device

- **exception** `gpiozero.BadEventHandler`
  
  Bases: `GPIOZeroError` (page 239), `ValueError`
  
  Error raised when an event handler with an incompatible prototype is specified

---

903: https://docs.python.org/3.9/library/exceptions.html#ValueError

904: https://docs.python.org/3.9/library/exceptions.html#ValueError

905: https://docs.python.org/3.9/library/exceptions.html#Exception

906: https://docs.python.org/3.9/library/exceptions.html#ValueError
exception gpiozero.BadWaitTime
    Bases: GPIOZeroError (page 239), ValueError
    Error raised when an invalid wait time is specified

exception gpiozero.BadQueueLen
    Bases: GPIOZeroError (page 239), ValueError
    Error raised when non-positive queue length is specified

exception gpiozero.BadPinFactory
    Bases: GPIOZeroError (page 239), ImportError
    Error raised when an unknown pin factory name is specified

exception gpiozero.ZombieThread
    Bases: GPIOZeroError (page 239), RuntimeError
    Error raised when a thread fails to die within a given timeout

exception gpiozero.CompositeDeviceError
    Bases: GPIOZeroError (page 239)
    Base class for errors specific to the CompositeDevice hierarchy

exception gpiozero.CompositeDeviceBadName
    Bases: CompositeDeviceError (page 240), ValueError
    Error raised when a composite device is constructed with a reserved name

exception gpiozero.CompositeDeviceBadOrder
    Bases: CompositeDeviceError (page 240), ValueError
    Error raised when a composite device is constructed with an incomplete order

exception gpiozero.CompositeDeviceBadDevice
    Bases: CompositeDeviceError (page 240), ValueError
    Error raised when a composite device is constructed with an object that doesn’t inherit from Device (page 199)

exception gpiozero.EnergenieSocketMissing
    Bases: CompositeDeviceError (page 240), ValueError
    Error raised when socket number is not specified

exception gpiozero.EnergenieBadSocket
    Bases: CompositeDeviceError (page 240), ValueError
    Error raised when an invalid socket number is passed to Energenie (page 180)

exception gpiozero.SPIError
    Bases: GPIOZeroError (page 239)
    Base class for errors related to the SPI implementation

---

907 https://docs.python.org/3.9/library/exceptions.html#ValueError
908 https://docs.python.org/3.9/library/exceptions.html#ValueError
909 https://docs.python.org/3.9/library/exceptions.html#ImportError
910 https://docs.python.org/3.9/library/exceptions.html#RuntimeError
911 https://docs.python.org/3.9/library/exceptions.html#ValueError
912 https://docs.python.org/3.9/library/exceptions.html#ValueError
913 https://docs.python.org/3.9/library/exceptions.html#ValueError
914 https://docs.python.org/3.9/library/exceptions.html#ValueError
915 https://docs.python.org/3.9/library/exceptions.html#ValueError
exception gpiozero.SPIBadArgs
   Bases: SPIError (page 240), ValueError
   Error raised when invalid arguments are given while constructing SPIDevice (page 154)

exception gpiozero.SPIBadChannel
   Bases: SPIError (page 240), ValueError
   Error raised when an invalid channel is given to an AnalogInputDevice (page 153)

exception gpiozero.SPIFixedClockMode
   Bases: SPIError (page 240), AttributeError
   Error raised when the SPI clock mode cannot be changed

exception gpiozero.SPIInvalidClockMode
   Bases: SPIError (page 240), ValueError
   Error raised when an invalid clock mode is given to an SPI implementation

exception gpiozero.SPIFixedBitOrder
   Bases: SPIError (page 240), AttributeError
   Error raised when the SPI bit-endianness cannot be changed

exception gpiozero.SPIFixedSelect
   Bases: SPIError (page 240), AttributeError
   Error raised when the SPI select polarity cannot be changed

exception gpiozero.SPIFixedWordSize
   Bases: SPIError (page 240), AttributeError
   Error raised when the number of bits per word cannot be changed

exception gpiozero.SPIInvalidWordSize
   Bases: SPIError (page 240), ValueError
   Error raised when an invalid (out of range) number of bits per word is specified

exception gpiozero.GPIODeviceError
   Bases: GPIOZeroError (page 239)
   Base class for errors specific to the GPIODevice hierarchy

exception gpiozero.GPIODeviceClosed
   Bases: GPIODeviceError (page 241), DeviceClosed (page 239)
   Deprecated descendent of DeviceClosed (page 239)

exception gpiozero.GPIOPinInUse
   Bases: GPIODeviceError (page 241)
   Error raised when attempting to use a pin already in use by another device

exception gpiozero.GPIOPinMissing
   Bases: GPIODeviceError (page 241), ValueError
   Error raised when a pin specification is not given

916 https://docs.python.org/3.9/library/exceptions.html#ValueError
917 https://docs.python.org/3.9/library/exceptions.html#ValueError
918 https://docs.python.org/3.9/library/exceptions.html#AttributeError
919 https://docs.python.org/3.9/library/exceptions.html#ValueError
920 https://docs.python.org/3.9/library/exceptions.html#AttributeError
921 https://docs.python.org/3.9/library/exceptions.html#AttributeError
922 https://docs.python.org/3.9/library/exceptions.html#AttributeError
923 https://docs.python.org/3.9/library/exceptions.html#ValueError
924 https://docs.python.org/3.9/library/exceptions.html#ValueError
exception gpiozero.InputDeviceError
    Bases: GPIODeviceError (page 241)
    Base class for errors specific to the InputDevice hierarchy

exception gpiozero.OutputDeviceError
    Bases: GPIODeviceError (page 241)
    Base class for errors specified to the OutputDevice hierarchy

exception gpiozero.OutputDeviceBadValue
    Bases: OutputDeviceError (page 242), ValueError
    Error raised when value is set to an invalid value

exception gpiozero.PinError
    Bases: GPIOZeroError (page 239)
    Base class for errors related to pin implementations

exception gpiozero.PinInvalidFunction
    Bases: PinError (page 242), ValueError
    Error raised when attempting to change the function of a pin to an invalid value

exception gpiozero.PinInvalidState
    Bases: PinError (page 242), ValueError
    Error raised when attempting to assign an invalid state to a pin

exception gpiozero.PinInvalidPull
    Bases: PinError (page 242), ValueError
    Error raised when attempting to assign an invalid pull-up to a pin

exception gpiozero.PinInvalidEdges
    Bases: PinError (page 242), ValueError
    Error raised when attempting to assign an invalid edge detection to a pin

exception gpiozero.PinInvalidBounce
    Bases: PinError (page 242), ValueError
    Error raised when attempting to assign an invalid bounce time to a pin

exception gpiozero.PinSetInput
    Bases: PinError (page 242), AttributeError
    Error raised when attempting to set a read-only pin

exception gpiozero.PinFixedPull
    Bases: PinError (page 242), AttributeError
    Error raised when attempting to set the pull of a pin with fixed pull-up

exception gpiozero.PinEdgeDetectUnsupported
    Bases: PinError (page 242), AttributeError
    Error raised when attempting to use edge detection on unsupported pins

---

925 https://docs.python.org/3.9/library/exceptions.html#ValueError
926 https://docs.python.org/3.9/library/exceptions.html#ValueError
927 https://docs.python.org/3.9/library/exceptions.html#ValueError
928 https://docs.python.org/3.9/library/exceptions.html#ValueError
929 https://docs.python.org/3.9/library/exceptions.html#ValueError
930 https://docs.python.org/3.9/library/exceptions.html#AttributeError
931 https://docs.python.org/3.9/library/exceptions.html#AttributeError
932 https://docs.python.org/3.9/library/exceptions.html#AttributeError
933 https://docs.python.org/3.9/library/exceptions.html#AttributeError
exception gpiozero.PinUnsupported
    Bases: PinError (page 242), NotImplementedError
    Error raised when attempting to obtain a pin interface on unsupported pins

exception gpiozero.PinSPIUnsupported
    Bases: PinError (page 242), NotImplementedError
    Error raised when attempting to obtain an SPI interface on unsupported pins

exception gpiozero.PinPWMError
    Bases: PinError (page 242)
    Base class for errors related to PWM implementations

exception gpiozero.PinPWMUnsupported
    Bases: PinPWMError (page 243), AttributeError
    Error raised when attempting to activate PWM on unsupported pins

exception gpiozero.PinPWMFixedValue
    Bases: PinPWMError (page 243), AttributeError
    Error raised when attempting to initialize PWM on an input pin

exception gpiozero.PinUnknownPi
    Bases: PinError (page 242), RuntimeError
    Error raised when gpiozero doesn’t recognize a revision of the Pi

exception gpiozero.PinMultiplePins
    Bases: PinError (page 242), RuntimeError
    Error raised when multiple pins support the requested function

exception gpiozero.PinNoPins
    Bases: PinError (page 242), RuntimeError
    Error raised when no pins support the requested function

exception gpiozero.PinInvalidPin
    Bases: PinError (page 242), 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

934 https://docs.python.org/3.9/library/exceptions.html#NotImplementedError
935 https://docs.python.org/3.9/library/exceptions.html#NotImplementedError
936 https://docs.python.org/3.9/library/exceptions.html#AttributeError
937 https://docs.python.org/3.9/library/exceptions.html#AttributeError
938 https://docs.python.org/3.9/library/exceptions.html#RuntimeError
939 https://docs.python.org/3.9/library/exceptions.html#RuntimeError
940 https://docs.python.org/3.9/library/exceptions.html#RuntimeError
941 https://docs.python.org/3.9/library/exceptions.html#ValueError
942 https://docs.python.org/3.9/library/exceptions.html#Warning
exception gpiozero.DistanceSensorNoEcho
    Bases: GPIOZeroWarning (page 243)
    Warning raised when the distance sensor sees no echo at all

exception gpiozero.SPIWarning
    Bases: GPIOZeroWarning (page 243)
    Base class for warnings related to the SPI implementation

exception gpiozero.SPISoftwareFallback
    Bases: SPIWarning (page 244)
    Warning raised when falling back to the SPI software implementation

exception gpiozero.PinWarning
    Bases: GPIOZeroWarning (page 243)
    Base class for warnings related to pin implementations

exception gpiozero.PinFactoryFallback
    Bases: PinWarning (page 244)
    Warning raised when a default pin factory fails to load and a fallback is tried

exception gpiozero.PinNonPhysical
    Bases: PinWarning (page 244)
    Warning raised when a non-physical pin is specified in a constructor

exception gpiozero.ThresholdOutOfRange
    Bases: GPIOZeroWarning (page 243)
    Warning raised when a threshold is out of range specified by min and max values

exception gpiozero.CallbackSetToNone
    Bases: GPIOZeroWarning (page 243)
    Warning raised when a callback is set to None when its previous value was None
26.1 Release 2.0 (2023-09-12)

- Removed Python 2.x support; many thanks to Fangchen Li for a substantial amount of work on this! (#799, #896)
- Removed RPIO pin implementation
- Made `gpiozero.pins.lgpio.LGPIOFactory` (page 235) the default factory; the former default, `gpiozero.pins.rpigpio.RPiGPIOFactory` (page 235), is now the second place preference
- Added Backwards Compatibility (page 87) chapter
- Added `pintest` utility
- Added Raspberry Pi 5 board data

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 115) class (thanks to Paulo Mateus) (#482, #928)
- Added support for multi-segment character displays with `LEDCharDisplay` (page 162) and `LEDMultiCharDisplay` (page 163) along with “font” support using `LEDCharFont` (page 165) (thanks to Martin

---

943 https://github.com/gpiozero/gpiozero/issues/799
944 https://github.com/gpiozero/gpiozero/issues/896
945 https://github.com/gpiozero/gpiozero/issues/482
946 https://github.com/gpiozero/gpiozero/issues/928
O’Hanlon) (#357, #485, #488, #493, #930)
- Added Pibrella (page 175) class (thanks to Carl Monk) (#773, #798)
- Added TrafficpHat (page 174) class (thanks to Ryan Walmsley) (#845, #846)
- Added support for the lgpio library as a pin factory (LGPIOFactor (page 235)) (thanks to Joan for lg) (#927)
- Allow Motor (page 134) to pass pin_factory (page 199) to its child OutputDevice (page 144) objects (thanks to Yisrael Dov Lebow) (#792)
- Small SPI exception fix (thanks to Maksim Levental) (#762)
- Warn users when using default pin factory for Servos and Distance Sensors (thanks to Sofia Kosovan and Daniele Procida at the EuroPythonsprints) (#780, #781)
- Added pulse_width (page 138) property to Servo (page 137) (suggested by Daniele Procida at the PyCon UK sprints) (#795, #796)
- Added event-driven functionality to internal devices (page 189) (#941)
- Allowed Energenie (page 180) sockets preserve their state on construction (thanks to Jack Wearden) (#865)
- Added source tools scaled_half() and scaled_full()
- Added complete Pi 4 support to NativeFactory (page 237) (thanks to Andrew Scheller) (#920, #929, #940)
- Updated add-on boards to use BOARD numbering (#349, #860)
- Fixed ButtonBoard (page 165) release events (#761)
- Add ASCII art diagrams to pinout for Pi 400 and CM4 (#932)
- Cleaned up software SPI (thanks to Andrew Scheller and Kyle Morgan) (#777, #895, #900)
- Added USB3 and Ethernet speed attributes to pi_info() (page 217)
- Various docs updates

947 https://github.com/gpiozero/gpiozero/issues/357
948 https://github.com/gpiozero/gpiozero/issues/485
949 https://github.com/gpiozero/gpiozero/issues/488
950 https://github.com/gpiozero/gpiozero/issues/493
951 https://github.com/gpiozero/gpiozero/issues/930
952 https://github.com/gpiozero/gpiozero/issues/773
953 https://github.com/gpiozero/gpiozero/issues/798
954 https://github.com/gpiozero/gpiozero/issues/845
955 https://github.com/gpiozero/gpiozero/issues/846
956 http://abyz.me.uk/lg/py_lgpio.html
957 https://github.com/gpiozero/gpiozero/issues/927
958 https://github.com/gpiozero/gpiozero/issues/792
959 https://github.com/gpiozero/gpiozero/issues/762
960 https://github.com/gpiozero/gpiozero/issues/780
961 https://github.com/gpiozero/gpiozero/issues/781
962 https://github.com/gpiozero/gpiozero/issues/795
963 https://github.com/gpiozero/gpiozero/issues/797
964 https://github.com/gpiozero/gpiozero/issues/941
965 https://github.com/gpiozero/gpiozero/issues/865
966 https://github.com/gpiozero/gpiozero/issues/920
967 https://github.com/gpiozero/gpiozero/issues/929
968 https://github.com/gpiozero/gpiozero/issues/940
969 https://github.com/gpiozero/gpiozero/issues/349
970 https://github.com/gpiozero/gpiozero/issues/860
971 https://github.com/gpiozero/gpiozero/issues/761
972 https://github.com/gpiozero/gpiozero/issues/932
973 https://github.com/gpiozero/gpiozero/issues/777
974 https://github.com/gpiozero/gpiozero/issues/895
975 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 217) 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 113). (#664976, #665977)
- Further improvements to `DistanceSensor` (page 113) (ignoring missed edges). (#719978)
- Allow source to take a device object as well as values or other values. See `Source/Values` (page 61). (#640979)
- Added internal device classes `LoadAverage` (page 193) and `DiskUsage` (page 195) (thanks to Jeevan M R for the latter). (#532980, #714981)
- Added support for `colorzero` with `RGBLED` (page 128) (this adds a new dependency). (#655983)
- Added `TonalBuzzer` (page 133) with `Tone` (page 215) API for specifying frequencies raw or via MIDI or musical notes. (#681984, #717985)
- Added `PiHutXmasTree` (page 169). (#502986)
- Added `PumpkinPi` (page 184) and `JamHat` (page 174) (thanks to Claire Pollard). (#680987, #681988, #717989)
- Ensured gpiozero can be imported without a valid pin factory set. (#591990, #713991)
- Reduced import time by not computing default pin factory at the point of import. (#675992, #722993)
- Added support for various pin numbering mechanisms. (#470994)
- `Motor` (page 134) instances now use `DigitalOutputDevice` (page 141) for non-PWM pins.
- Allow non-PWM use of `Robot` (page 176). (#481995)
- Added optional `enable` init param to `Motor` (page 134). (#366996)
- Added `--xyz` option to `pinout` command line tool to open `pinout.xyz` in a web browser. (#604998)
• Added 3B+, 3A+ and CM3+ to Pi model data. (#627999, #7041000)
• Minor improvements to *Energenie* (page 180), thanks to Steve Amor. (#6291001, #6341002)
• Allow *SmoothedInputDevice* (page 120), *LightSensor* (page 111) and *MotionSensor* (page 109) to have pull-up configured. (#6521003)
• Allow input devices to be pulled up or down externally, thanks to Philippe Muller. (#5931004, #6581005)
• Minor changes to support Python 3.7, thanks to Russel Winder and Rick Ansell. (#6661006, #6681007, #6691008, #6711009, #6731010)
• Added *zip_values()* (page 208) source tool.
• Correct row/col numbering logic in *PinInfo* (page 219). (#6741011)
• 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 177) and *Robot.backward()* (page 177). (#3061012 and #6191013)
• Fixed *DistanceSensor* (page 113) returning incorrect readings after a long pause, and added a lock to ensure multiple distance sensors can operate simultaneously in a single project. (#5841014, #5951015, #6171016, #6181017)
• Added support for phase/enable motor drivers with *PhaseEnableMotor* (page 136), *PhaseEnableRobot* (page 178), and descendants, thanks to Ian Harcombe! (#3861018)
• A variety of other minor enhancements, largely thanks to Andrew Scheller! (#4791019, #4891020, #4911021, #4921022)

---

999 https://github.com/gpiozero/gpiozero/issues/627
1000 https://github.com/gpiozero/gpiozero/issues/704
1001 https://github.com/gpiozero/gpiozero/issues/629
1002 https://github.com/gpiozero/gpiozero/issues/634
1003 https://github.com/gpiozero/gpiozero/issues/652
1004 https://github.com/gpiozero/gpiozero/issues/593
1005 https://github.com/gpiozero/gpiozero/issues/658
1006 https://github.com/gpiozero/gpiozero/issues/666
1007 https://github.com/gpiozero/gpiozero/issues/668
1008 https://github.com/gpiozero/gpiozero/issues/669
1009 https://github.com/gpiozero/gpiozero/issues/671
1010 https://github.com/gpiozero/gpiozero/issues/673
1011 https://github.com/gpiozero/gpiozero/issues/674
1012 https://github.com/gpiozero/gpiozero/issues/306
1013 https://github.com/gpiozero/gpiozero/issues/619
1014 https://github.com/gpiozero/gpiozero/issues/584
1015 https://github.com/gpiozero/gpiozero/issues/595
1016 https://github.com/gpiozero/gpiozero/issues/617
1017 https://github.com/gpiozero/gpiozero/issues/618
1018 https://github.com/gpiozero/gpiozero/issues/386
1019 https://github.com/gpiozero/gpiozero/issues/479
1020 https://github.com/gpiozero/gpiozero/issues/489
1021 https://github.com/gpiozero/gpiozero/issues/491
1022 https://github.com/gpiozero/gpiozero/issues/492
26.8 Release 1.4.0 (2017-07-26)

- Pin factory is now configurable from device constructors (page 223) as well as command line. NOTE: this is a backwards incompatible change for manual pin construction but it’s hoped this is (currently) a sufficiently rare use case that this won’t affect too many people and the benefits of the new system warrant such a change, i.e. the ability to use remote pin factories with HAT classes that don’t accept pin assignations (#2791023)
- Major work on SPI, primarily to support remote hardware SPI (#4211024, #4591025, #4651026, #4681027, #5751028)
- Pin reservation now works properly between GPIO and SPI devices (#4591029, #4681030)
- Lots of work on the documentation: source/values chapter (page 61), better charts, more recipes, remote GPIO configuration (page 45), mock pins, better PDF output (#4841031, #4691032, #5231033, #5201034, #4341035, #5651036, #5761037)
- Support for StatusZero (page 181) and StatusBoard (page 182) HATs (#5581038)
- Added pinout command line tool to provide a simple reference to the GPIO layout and information about the associated Pi (#4971039, #5041040) thanks to Stewart Adcock for the initial work
- Fixed a variety of packaging issues (#5351042, #5181043, #5191044)
- Improved text in factory fallback warnings (#5721045)

26.9 Release 1.3.2 (2017-03-03)

- Added new Pi models to stop pi_info() (page 217) breaking
- Fix issue with pi_info() (page 217) 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 165) for reading multiple buttons in a single class (#340)
- Added `Servo` (page 137) and `AngularServo` (page 139) classes for controlling simple servo motors (#248)
- Lots of work on supporting easier use of internal and third-party pin implementations (#359)
- `Robot` (page 176) now has a proper `value` (page 178) attribute (#305)
- Added `CPUTemperature` (page 192) as another demo of “internal” devices (#294)
- A temporary work-around for an issue with `DistanceSensor` (page 113) 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 180) class for controlling Energenie plugs (#69)
- Added `LineSensor` (page 108) class for single line-sensors (#109)
- Added `DistanceSensor` (page 113) class for HC-SR04 ultra-sonic sensors (#114)
- Added `SnowPi` (page 183) class for the Ryanteck Snow-pi board (#130)
- Added `when_held` (page 106) (and related properties) to `Button` (page 105) (#115)
- Fixed issues with installing GPIO Zero for python 3 on Raspbian Wheezy releases (#140)
- Added support for lots of ADC chips (MCP3xx family) (#162) - many thanks to pocpa and lurch!
- Added support for pigpiod as a pin implementation with `PiGPIOPin` (page 236) (#180)

---

1046 https://github.com/gpiozero/gpiozero/issues/340
1047 https://github.com/gpiozero/gpiozero/issues/248
1048 https://github.com/gpiozero/gpiozero/issues/359
1049 https://github.com/gpiozero/gpiozero/issues/305
1050 https://github.com/gpiozero/gpiozero/issues/294
1051 https://github.com/gpiozero/gpiozero/issues/385
1052 https://github.com/gpiozero/gpiozero/issues/320
1053 https://github.com/gpiozero/gpiozero/issues/295
1054 https://github.com/gpiozero/gpiozero/issues/289
1055 https://github.com/gpiozero/gpiozero/issues/69
1056 https://github.com/gpiozero/gpiozero/issues/109
1057 https://github.com/gpiozero/gpiozero/issues/114
1058 https://github.com/gpiozero/gpiozero/issues/130
1059 https://github.com/gpiozero/gpiozero/issues/115
1060 https://github.com/gpiozero/gpiozero/issues/140
1061 https://github.com/gpiozero/gpiozero/issues/162
1062 https://github.com/gpiozero/gpiozero/issues/180
Many refinements to the base classes mean more consistency in composite devices and several bugs squashed (#164, #175, #182, #189, #193, #229)

GPIO Zero is now aware of what sort of Pi it’s running on via `pi_info()` (page 217) and has a fairly extensive database of Pi information which it uses to determine when users request impossible things (like pull-down on a pin with a physical pull-up resistor) (#222)

The source/values system was enhanced to ensure normal usage doesn’t stress the CPU and lots of utilities were added (#181, #251)

And I’ll just add a note of thanks to the many people in the community who contributed to this release: we’ve had some great PRs, suggestions, and bug reports in this version. Of particular note:

- Schelt 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!

### 26.13 Release 1.1.0 (2016-02-08)

- Documentation converted to reST and expanded to include generic classes and several more recipes (#80, #82, #101, #119, #135, #168)
- New `CamJamKitRobot` (page 179) class with the pre-defined motor pins for the new CamJam EduKit
- New `LEDBarGraph` (page 160) class (many thanks to Martin O’Hanlon!) (#126, #176)
- New `Pin` (page 227) implementation abstracts out the concept of a GPIO pin paving the way for alternate library support and IO extenders in future (#141)
- New `LEDBoard.blink()` (page 158) method which works properly even when background is set to `False` (#94, #161)
- New `RGBLED.blink()` (page 129) method which implements (rudimentary) color fading too! (#135, #174)
- New `initial_value` attribute on `OutputDevice` (page 144) ensures consistent behaviour on construction (#118)
• New `active_high` attribute on `PWMOutputDevice` (page 142) and `RGBLED` (page 128) allows use of common anode devices (#143, #154)

• Loads of new ADC chips supported (many thanks to GitHub user pcopa!) (#150)

### 26.14 Release 1.0.0 (2015-11-16)

• Debian packaging added (#44)

• `PWMLED` (page 127) class added (#58)

• `TemperatureSensor` removed pending further work (#93)

• `Buzzer.beep()` (page 132) alias method added (#75)

• `Motor` (page 134) PWM devices exposed, and `Robot` (page 176) motor devices exposed (#107)

### 26.15 Release 0.9.0 (2015-10-25)

Fourth public beta

• Added source and values properties to all relevant classes (#76)

• Fix names of parameters in `Motor` (page 134) constructor (#79)

• Added wrappers for LED groups on add-on boards (#81)

### 26.16 Release 0.8.0 (2015-10-16)

Third public beta

• Added generic `AnalogInputDevice` (page 153) class along with specific classes for the `MCP3008` (page 149) and `MCP3004` (page 149) (#41)

• Fixed `DigitalOutputDevice.blink()` (page 141) (#57)

---

1086 https://github.com/gpiozero/gpiozero/issues/143
1087 https://github.com/gpiozero/gpiozero/issues/154
1088 https://github.com/gpiozero/gpiozero/issues/150
1089 https://github.com/gpiozero/gpiozero/issues/44
1090 https://github.com/gpiozero/gpiozero/issues/58
1091 https://github.com/gpiozero/gpiozero/issues/93
1092 https://github.com/gpiozero/gpiozero/issues/75
1093 https://github.com/gpiozero/gpiozero/issues/107
1094 https://github.com/gpiozero/gpiozero/issues/76
1095 https://github.com/gpiozero/gpiozero/issues/79
1096 https://github.com/gpiozero/gpiozero/issues/81
1097 https://github.com/gpiozero/gpiozero/issues/41
1098 https://github.com/gpiozero/gpiozero/issues/57
26.17 Release 0.7.0 (2015-10-09)

Second public beta

26.18 Release 0.6.0 (2015-09-28)

First public beta

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.
g

gpiozero, 3

gpiozero.boards, 157

gpiozero.devices, 197

gpiozero.exc, 239

gpiozero.fonts, 211

gpiozero.input_devices, 105

gpiozero.internal_devices, 189

gpiozero.output_devices, 125

gpiozero.pins, 221

gpiozero.pins.data, 217

gpiozero.pins.lgpio, 235

gpiozero.pins.local, 234

gpiozero.pins.mock, 237

gpiozero.pins.native, 237

gpiozero.pins.pi, 233

gpiozero.pins.pigpio, 236

gpiozero.pins.rpigpio, 235

gpiozero.spi_devices, 147

gpiozero.tones, 215

gpiozero.tools, 203
Symbols

_absolute_key()  (gpiozero.SharedMixin class method), 200
--color
  pinout command line option, 71
--help
  pinout command line option, 71
  pintest command line option, 76
--monochrome
  pinout command line option, 71
--pins
  pintest command line option, 76
--revision
  pinout command line option, 71
  pintest command line option, 76
--skip
  pintest command line option, 76
--version
  pintest command line option, 76
--xyz
  pinout command line option, 71
-c
  pinout command line option, 71
-h
  pinout command line option, 71
-m
  pinout command line option, 71
-p
  pintest command line option, 76
-r
  pinout command line option, 71
  pintest command line option, 76
-s
  pintest command line option, 76
-x
  pinout command line option, 71
-y
  pintest command line option, 76

A
absoluted()  (in module gpiozero.tools), 203
active_high (gpiozero.OutputDevice property), 144
active_time (gpiozero.DigitalInputDevice property), 119
active_time (gpiozero.EventsMixin property), 201
all_values()  (in module gpiozero.tools), 206
alternating_values()  (in module gpiozero.tools), 208
amber (gpiozero.TrafficLights attribute), 168
AnalogInputDevice (class in gpiozero), 153
angle (gpiozero.AngularServo property), 140
AngularServo (class in gpiozero), 139
any_values()  (in module gpiozero.tools), 207
arms (gpiozero.SnowPi attribute), 183
averaged()  (in module gpiozero.tools), 207

B
backward()  (gpiozero.Motor method), 135
backward()  (gpiozero.PhaseEnableMotor method), 136
backward()  (gpiozero.Robot method), 177
BadEventHandler, 239
BadPinFactory, 240
BadQueueLen, 240
BadWaitTime, 239
beep()  (gpiozero.Buzzer method), 132
bits (gpiozero.AnalogInputDevice property), 154
bits_per_word (gpiozero.SPI property), 230
blink()  (gpiozero.DigitalOutputDevice method), 141
blink()  (gpiozero.LED method), 126
blink()  (gpiozero.LEDBoard method), 158
blink()  (gpiozero.PWMLED method), 127
blink()  (gpiozero.PWMOutputDevice method), 142
blink()  (gpiozero.RGBLED method), 129
blue (gpiozero.RGBLED property), 130
board_info (gpiozero.Factory property), 227
booleanized()  (in module gpiozero.tools), 203
bounce (gpiozero.Pin property), 228
Button (class in gpiozero), 105
button (gpiozero.Pibrella attribute), 176
button (gpiozero.StatusBoard attribute), 182
button (gpiozero.TrafficLightsBuzzer attribute), 169
ButtonBoard (class in gpiozero), 165
Buzzer (class in gpiozero), 131
buzzer (gpiozero.JamHat attribute), 175
buzzer (gpiozero.Pibrella attribute), 176
buzzer (gpiozero.TrafficLightsBuzzer attribute), 168
callback_set_to_none, 244
CamJamKitRobot (class in gpiozero), 179
channel (gpiozero.MCP3002 property), 148
channel (gpiozero.MCP3004 property), 149
channel (gpiozero.MCP3008 property), 149
channel (gpiozero.MCP2002 property), 150
channel (gpiozero.MCP2004 property), 150
channel (gpiozero.MCP3008 property), 150
channel (gpiozero.MCP3002 property), 151
channel (gpiozero.MCP3004 property), 152
camped() (in module gpiozero.tools), 204
clock_mode (gpiozero.SPI property), 231
clock_phase (gpiozero.SPI property), 231
clock_polarity (gpiozero.SPI property), 231
close() (gpiozero.CompositeDevice method), 187
close() (gpiozero.Device method), 199
close() (gpiozero.Factory method), 226
close() (gpiozero.GPIODevice method), 122
close() (gpiozero.Pin method), 227
close() (gpiozero.pins.pi.PiFactoy method), 233
close() (gpiozero.SPIDevice method), 154
closed (gpiozero.CompositeDevice property), 188
closed (gpiozero.Device property), 199
closed (gpiozero.GPIODevice property), 123
closed (gpiozero.SPIDevice property), 155
col (gpiozero.PinInfo attribute), 219
color (gpiozero.RGBLED property), 130
columns (gpiozero.HeaderInfo attribute), 218
CompositeDevice (class in gpiozero), 187
CompositeDeviceBadDevice, 240
CompositeDeviceBadName, 240
CompositeDeviceBadOrder, 240
CompositeDeviceError, 240
CompositeOutputDevice (class in gpiozero), 186
cos_values() (in module gpiozero.tools), 208
CPUtemperature (class in gpiozero), 192
detach() (gpiozero.Servo method), 138
Device (class in gpiozero), 199
DeviceClosed, 239
differential (gpiozero.MCP3002 property), 148
differential (gpiozero.MCP3004 property), 149
differential (gpiozero.MCP3008 property), 149
differential (gpiozero.MCP2002 property), 150
differential (gpiozero.MCP2004 property), 150
differential (gpiozero.MCP3008 property), 150
differential (gpiozero.MCP3002 property), 151
differential (gpiozero.MCP3004 property), 152
DigitalInputDevice (class in gpiozero), 119
DigitalOutputDevice (class in gpiozero), 141
DiskUsage (class in gpiozero), 195
distance (gpiozero.DistanceSensor property), 114
DistanceSensor (class in gpiozero), 113
DistanceSensorNoEcho, 243
down() (gpiozero.tones.Tone method), 215
load_segment_font() (in module gpiozero.fonts), 212
 LOADAverage (class in gpiozero), 193
 LocalPiFactory (class in gpiozero.pins.local), 234
 LocalPiPin (class in gpiozero.pins.local), 235
 lsb_first (gpiozero.SPI property), 232

 M
 max() (gpiozero.AngularServo method), 140
 max() (gpiozero.Servo method), 138
 max_angle (gpiozero.AngularServo property), 140
 max_distance (gpiozero.DistanceSensor property), 114
 max_pulse_width (gpiozero.Servo property), 138
 max_steps (gpiozero.RotaryEncoder property), 116
 max_tone (gpiozero.TonalBuzzer property), 134
 max_voltage (gpiozero.AnalogInputDevice property), 154
 MCP3001 (class in gpiozero), 148
 MCP3002 (class in gpiozero), 148
 MCP3004 (class in gpiozero), 149
 MCP3008 (class in gpiozero), 149
 MCP3201 (class in gpiozero), 149
 MCP3202 (class in gpiozero), 150
 MCP3204 (class in gpiozero), 150
 MCP3208 (class in gpiozero), 150
 MCP3301 (class in gpiozero), 151
 MCP3302 (class in gpiozero), 151
 MCP3304 (class in gpiozero), 152
 mid() (gpiozero.AngularServo method), 140
 mid() (gpiozero.Servo method), 138
 mid_tone (gpiozero.TonalBuzzer property), 134
 midi (gpiozero.tones.Tone property), 216
 min() (gpiozero.AngularServo method), 140
 min() (gpiozero.Servo method), 138
 min_angle (gpiozero.AngularServo property), 140
 min_pulse_width (gpiozero.Servo property), 138
 min_tone (gpiozero.TonalBuzzer property), 134
 MockChargingPin (class in gpiozero.pins.mock), 238
 MockConnectedPin (class in gpiozero.pins.mock), 238
 MockFactory (class in gpiozero.pins.mock), 237
 MockPin (class in gpiozero.pins.mock), 238
 MockPIMPin (class in gpiozero.pins.mock), 238
 MockSPIDevice (class in gpiozero.pins.mock), 238
 MockTriggerPin (class in gpiozero.pins.mock), 238

gpiozero.pins.local, 234
 gpiozero.pins.mock, 237
 gpiozero.pins.native, 237
 gpiozero.pins.pl, 233
 gpiozero.pins.pigpio, 236
 gpiozero.pins.rpigpio, 235
 gpiozero.spi_devices, 147
 gpiozero.tones, 215
 gpiozero.tools, 203
 motion_detected (gpiozero.MotionSensor property), 110
 MotionSensor (class in gpiozero), 109
 Motor (class in gpiozero), 134
 multiplied() (in module gpiozero.tools), 207

 N
 name (gpiozero.HeaderInfo attribute), 218
 name (gpiozero.PinInfo attribute), 219
 namedtuple (gpiozero.CompositeDevice property), 188
 names (gpiozero.PinInfo attribute), 219
 Native2711Pin (class in gpiozero.pins.native), 237
 Native2835Pin (class in gpiozero.pins.native), 237
 NativeFactory (class in gpiozero.pins.native), 237
 NativePin (class in gpiozero.pins.native), 237
 negated() (in module gpiozero.tools), 204
 nose (gpiozero.SnowPi attribute), 183
 note (gpiozero.tones.Tone property), 216
 number (gpiozero.PinInfo attribute), 219

 O
 octaves (gpiozero.TonalBuzzer property), 134
 off() (gpiozero.Buzzer method), 132
 off() (gpiozero.CompositeOutputDevice method), 186
 off() (gpiozero.DigitalOutputDevice method), 142
 off() (gpiozero.EnergyGpio method), 180
 off() (gpiozero.JamHat method), 175
 off() (gpiozero.LED method), 126
 off() (gpiozero.LEDBoard method), 158
 off() (gpiozero.OutputDevice method), 144
 off() (gpiozero.Pibrella method), 176
 off() (gpiozero.PWMLED method), 127
 off() (gpiozero.PWMOutputDevice method), 143
 off() (gpiozero.RGBLED method), 130
 on() (gpiozero.Buzzer method), 132
 on() (gpiozero.CompositeOutputDevice method), 186
 on() (gpiozero.DigitalOutputDevice method), 142
 on() (gpiozero.EnergyGpio method), 180
 on() (gpiozero.JamHat method), 175
 on() (gpiozero.LED method), 126
 on() (gpiozero.LEDBoard method), 158
 on() (gpiozero.OutputDevice method), 144
 on() (gpiozero.Pibrella method), 176
 on() (gpiozero.PWMLED method), 127
 on() (gpiozero.PWMOutputDevice method), 143
 on() (gpiozero.RGBLED method), 130
 output_with_state() (gpiozero.Pin method), 228
 OutputDevice (class in gpiozero), 144
OutputDeviceBadValue, 242
OutputDeviceError, 242
outputs (gpiozero.Pinbrella attribute), 176

P

partial (gpiozero.SmoothedInputDevice property), 121
PhaseEnableMotor (class in gpiozero), 136
PhaseEnableRobot (class in gpiozero), 178
pi_info () (in module gpiozero), 217
PiBoardInfo (class in gpiozero), 217
Pibrella (class in gpiozero), 175
PiFactory (class in gpiozero.pins.pi), 233
PIGPIO_ADDR, 48, 58, 224
Pin (class in gpiozero.pins.pigpio), 236
PinGPIOFactory (class in gpiozero.pins.pigpio), 236
PiHutXmasTree (class in gpiozero), 169
PiLitter (class in gpiozero), 170
PiLightBarGraph (class in gpiozero), 171
Pin (class in gpiozero), 227
pin (gpiozero.Button property), 106
pin (gpiozero.Buzzer property), 132
pin (gpiozero.GPIODevice property), 123
pin (gpiozero.LED property), 126
pin (gpiozero.LightSensor property), 112
pin (gpiozero.LineSensor property), 108
pin (gpiozero.MotionSensor property), 110
pin (gpiozero.PWMLED property), 128
pin () (gpiozero.Factory method), 226
pin() (gpiozero.pins.pi.MockFactory method), 237
pin() (gpiozero.pins.pi.PinFactory method), 233
pin_class (gpiozero.pins.mock.MockFactory attribute), 237
pin_factory (gpiozero.Factory attribute), 199
PinEdgeDetectUnsupported, 242
PinError, 242
PinFactoryFallback, 244
PinFixedPull, 242
PingServer (class in gpiozero), 191
PinInfo (class in gpiozero), 219
PinInvalidBounce, 242
PinInvalidEdges, 242
PinInvalidFunction, 242
PinInvalidPin, 243
PinInvalidPull, 242
PinInvalidState, 242
PinMultiplePins, 243
PinNonPhysical, 244
PinNoPins, 243
pinout command line option
--color, 71
--help, 71
--monochrome, 71
--revision, 71
--xyz, 71
-c, 71
-h, 71
-m, 71

PinPWMError, 243
PinPWMFixedValue, 243
PinPWMUnsupported, 243
pins (gpiozero.HeaderInfo attribute), 218
PinSetInput, 242
PinSPIUnsupported, 243
pintest command line option
--help, 76
--pins, 76
--revision, 76
--skip, 76
--version, 76
--yes, 76
-h, 76
-p, 76
-r, 76
-s, 76
-y, 76
PinUnknownPi, 243
PinUnsupported, 242
PinWarning, 244
PiPin (class in gpiozero.pins.pi), 234
PiStop (class in gpiozero), 172
PiTraffic (class in gpiozero), 171
play () (gpiozero.TonalBuzzer method), 133
plex_delay (gpiozero.LEDMultiCharDisplay property), 164
PolledInternalDevice (class in gpiozero), 196
PololuDRV8835Robot (class in gpiozero), 179
post_delayed () (in module gpiozero.tools), 204
post_periodic_filtered () (in module gpiozero.tools), 204
pprint () (gpiozero.HeaderInfo method), 218
pre_delayed () (in module gpiozero.tools), 205
pre_periodic_filtered () (in module gpiozero.tools), 205
pressed_time (gpiozero.ButtonBoard property), 166
pull (gpiozero.Pin property), 229
pull (gpiozero.PinInfo attribute), 219
pull_up (gpiozero.Button property), 106
pull_up (gpiozero.InputDevice property), 122
pulse () (gpiozero.LEDBoard method), 159
pulse () (gpiozero.PWMLED method), 127
pulse () (gpiozero.PWMOutputDevice method), 143
pulse () (gpiozero.RGBLED method), 130
pulse_width (gpiozero.Servo property), 138
PumpkinPi (class in gpiozero), 184
PWMLED (class in gpiozero), 127
PWMOutputDevice (class in gpiozero), 142
quantized () (in module gpiozero.tools), 205
queue_len (gpiozero.SmoothedInputDevice property), 121
queued () (in module gpiozero.tools), 205

Index 263
SPIInvalidClockMode
SPIFixedSelect
SPIFixedClockMode

...
wait_for_active() (gpiozero.EventsMixin method), 201
wait_for_dark() (gpiozero.LightSensor method), 112
wait_for_in_range() (gpiozero.DistanceSensor method), 114
wait_for_inactive() (gpiozero.DigitalInputDevice method), 119
wait_for_inactive() (gpiozero.EventsMixin method), 201
wait_for_light() (gpiozero.LightSensor method), 112
wait_for_line() (gpiozero.LineSensor method), 108
wait_for_motion() (gpiozero.MotionSensor method), 110
wait_for_no_line() (gpiozero.LineSensor method), 108
wait_for_no_motion() (gpiozero.MotionSensor method), 110
wait_for_out_of_range() (gpiozero.DistanceSensor method), 114
wait_for_press() (gpiozero.Button method), 106
wait_for_press() (gpiozero.ButtonBoard method), 166
wait_for_release() (gpiozero.Button method), 106
wait_for_release() (gpiozero.ButtonBoard method), 166
wait_for_rotate() (gpiozero.RotaryEncoder method), 116
wait_for_rotate_clockwise() (gpiozero.RotaryEncoder method), 116
wait_for_rotate_counter_clockwise() (gpiozero.RotaryEncoder method), 116
when_activated (gpiozero.CPUTemperature attribute), 193
when_activated (gpiozero.DigitalInputDevice attribute), 119
when_activated (gpiozero.DiskUsage attribute), 195
when_activated (gpiozero.EventsMixin attribute), 201
when_activated (gpiozero.LoadAverage attribute), 194
when_activated (gpiozero.Pin attribute), 192
when_activated (gpiozero.TimeOfDay attribute), 190
when_changed (gpiozero.Pin property), 229
when_dark (gpiozero.LightSensor attribute), 112
when_deactivated (gpiozero.CPUTemperature attribute), 193
when_deactivated (gpiozero.DigitalInputDevice attribute), 119
when_deactivated (gpiozero.DiskUsage attribute), 195
when_deactivated (gpiozero.EventsMixin attribute), 201
when_deactivated (gpiozero.LoadAverage attribute), 194
when_deactivated (gpiozero.PingServer attribute), 192
when_deactivated (gpiozero.TimeOfDay attribute), 191
when_held (gpiozero.Button attribute), 106
when_held (gpiozero.HoldMixin attribute), 202
when_in_range (gpiozero.DistanceSensor attribute), 115
when_light (gpiozero.LightSensor attribute), 112
when_line (gpiozero.LineSensor attribute), 109
when_motion (gpiozero.MotionSensor attribute), 110
when_no_line (gpiozero.LineSensor attribute), 109
when_no_motion (gpiozero.MotionSensor attribute), 111
when_out_of_range (gpiozero.DistanceSensor attribute), 115
when_pressed (gpiozero.Button attribute), 107
when_pressed (gpiozero.ButtonBoard attribute), 166
when_released (gpiozero.Button attribute), 107
when_released (gpiozero.ButtonBoard attribute), 167
when_rotated (gpiozero.RotaryEncoder attribute), 117
when_rotated_clockwise (gpiozero.RotaryEncoder attribute), 117
when_rotated_counter_clockwise (gpiozero.RotaryEncoder attribute), 117
wrap (gpiozero.RotaryEncoder property), 117
write() (gpiozero.SPI method), 230

Y
yellow (gpiozero.TrafficLights attribute), 168

Z
zip_values() (in module gpiozero.tools), 208
ZombieThread, 240