Gpiozero Documentation

Release 1.1.0

Ben Nuttall

March 17, 2016

Contents

A simple interface to everyday GPIO components used with Raspberry Pi.

Created by Ben Nuttall of the Raspberry Pi Foundation, Dave Jones, and other contributors.

CHAPTER 1

About

Component interfaces are provided to allow a frictionless way to get started with physical computing:

```
from gpiozero import LED
from time import sleep

led = LED(17)

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

With very little code, you can quickly get going connecting your components together:

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

led = LED(17)
button = Button(3)

button.when_pressed = led.on
button.when_released = led.off

pause()
```

The library includes interfaces to many simple everyday components, as well as some more complex things like sensors, analogue-to-digital converters, full colour LEDs, robotics kits and more.

CHAPTER 2

Install

First, update your repositories list:

```
sudo apt-get update
```

Then install the package of your choice. Both Python 3 and Python 2 are supported. Python 3 is recommended:

```
sudo apt-get install python3-gpiozero
```

or:

```
sudo apt-get install python-gpiozero
```

CHAPTER 3

Documentation

Comprehensive documentation is available at https://gpiozero.readthedocs.org/.

CHAPTER 4

Development

This project is being developed on GitHub. Join in:

- Provide suggestions, report bugs and ask questions as issues
- Provide examples we can use as recipes
- Contribute to the code

Alternatively, email suggestions and feedback to mailto:ben@raspberrypi.org or add to the Google Doc.

CHAPTER 5

Contributors

- Ben Nuttall (project maintainer)
- Dave Jones
- Martin O'Hanlon

CHAPTER 6

Table of Contents

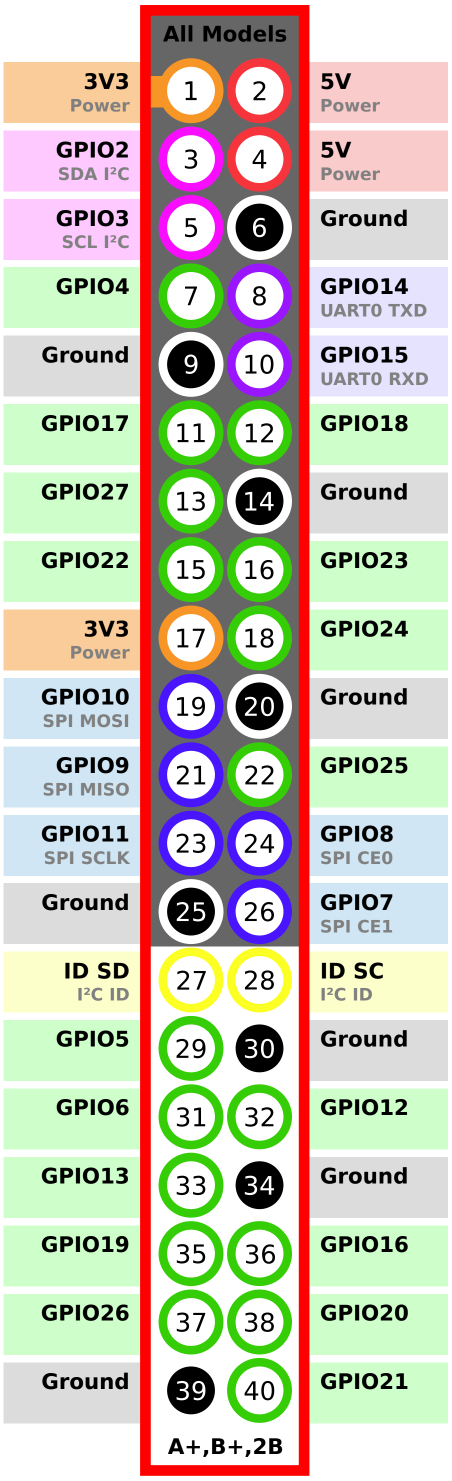
6.1 Recipes

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

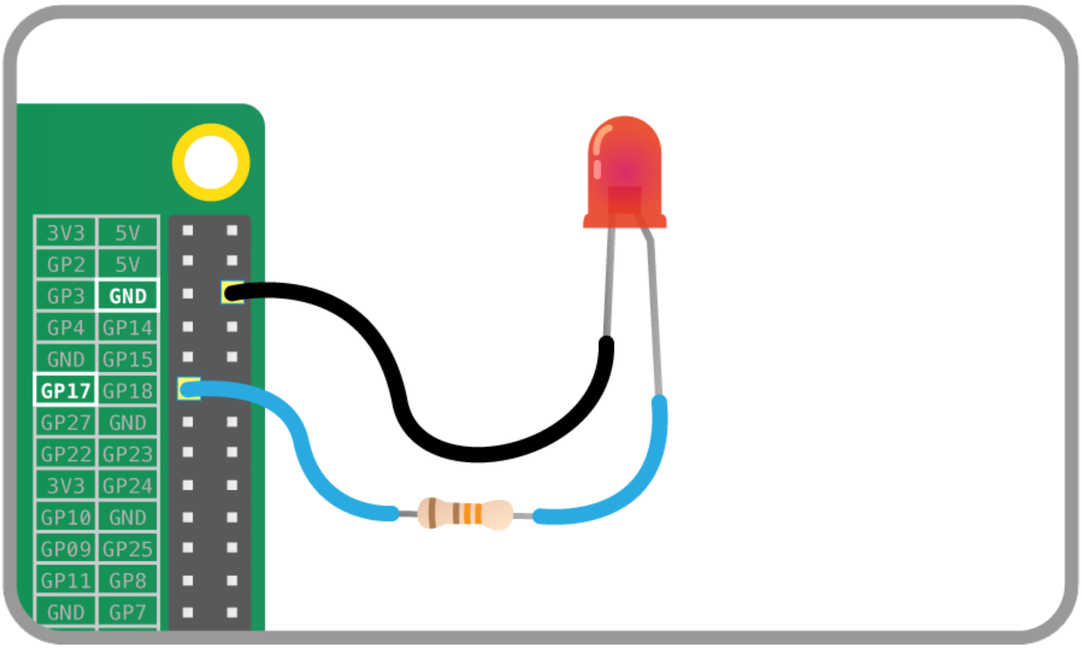
6.1.1 Pin Numbering

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.

Any pin marked `GPIO` in the diagram below can be used for generic components:


All Models
3V3 Power
1
2
5V Power
GPIO2 SDA I²C
3
4
5V Power
GPIO3 SCL I²C
5
6
Ground
GPIO4
7
8
GPIO14 UART0 TXD
Ground
9
10
GPIO15 UART0 RXD
GPIO17
11
12
GPIO18
GPIO27
13
14
Ground
GPIO22
15
16
GPIO23
3V3 Power
17
18
GPIO24
GPIO10 SPI MOSI
19
20
Ground
GPIO9 SPI MISO
21
22
GPIO25
GPIO11 SPI SCLK
23
24
GPIO8 SPI CE0
Ground
25
26
GPIO7 SPI CE1
ID SD I²C ID
27
28
ID SC I²C ID
GPIO5
29
30
Ground
GPIO6
31
32
GPIO12
GPIO13
33
34
Ground
GPIO19
35
36
GPIO16
GPIO26
37
38
GPIO20
Ground
39
40
GPIO21
A+,B+,2B

6.1.2 LED



Turn an *LED* on and off repeatedly:

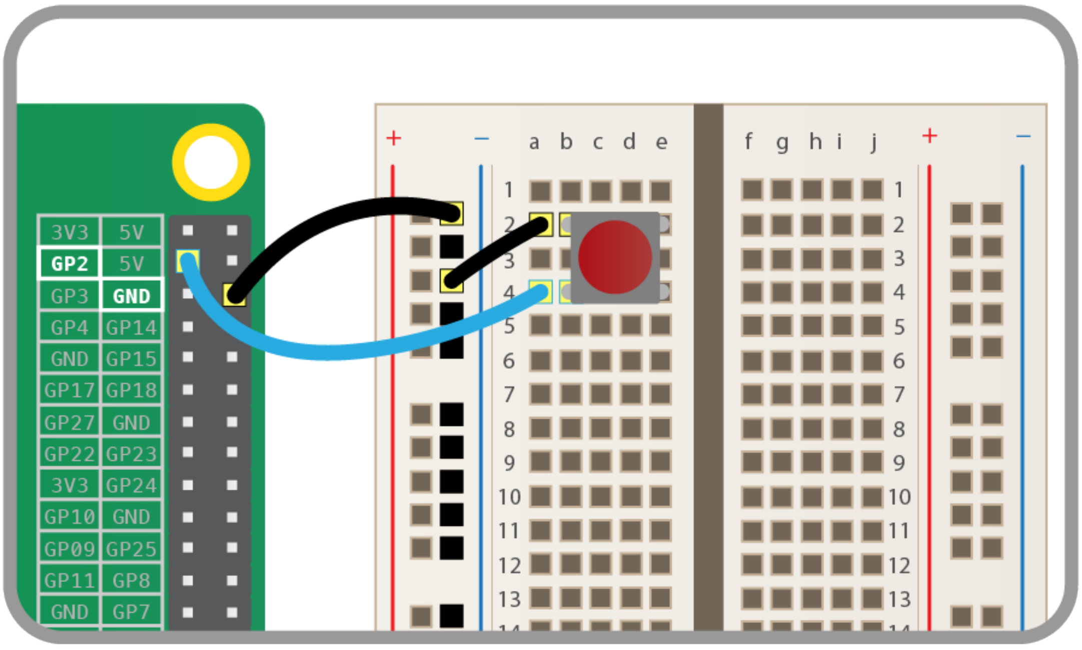
```
from gpiozero import LED
from time import sleep

red = LED(17)

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

Alternatively:

```
from gpiozero import LED
from signal import pause

red = LED(17)

red.blink()

pause()
```

Note: Reaching the end of a Python script will terminate the process and GPIOs may be reset. Keep your script alive with `signal.pause()`. See *Keep your script running* for more information.

6.1.3 Button



Check if a *Button* is pressed:

```
from gpiozero import Button

button = Button(2)

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

Wait for a button to be pressed before continuing:

```
from gpiozero import Button

button = Button(2)

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

Run a function every time the button is pressed:

```
from gpiozero import Button
from signal import pause

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

button = Button(2)

button.when_pressed = say_hello

pause()
```

6.1.4 Button controlled LED

Turn on an *LED* when a *Button* is pressed:

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

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

button.when_pressed = led.on
button.when_released = led.off

pause()
```

Alternatively:

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

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

led.source = button.values

pause()
```

6.1.5 Traffic Lights

A full traffic lights system.

Using a `TrafficLights` kit like Pi-Stop:

```
from gpiozero import TrafficLights
from time import sleep

lights = TrafficLights(2, 3, 4)

lights.green.on()

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

Alternatively:

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

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* components:

```
from gpiozero import LED
from time import sleep

red = LED(2)
amber = LED(3)
green = LED(4)

green.on()
amber.off()
red.off()

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

6.1.6 Push button stop motion

Capture a picture with the camera module every time a button is pressed:

```
from gpiozero import Button
from picamera import PiCamera

button = Button(2)

with PiCamera() as camera:
    camera.start_preview()
    frame = 1
    while True:
        button.wait_for_press()
        camera.capture('/home/pi/frame%03d.jpg' % frame)
        frame += 1
```

See Push Button Stop Motion for a full resource.

6.1.7 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 for a full resource.

6.1.8 GPIO Music Box

Each button plays a different sound!

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

pygame.mixer.init()

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

buttons = [Button(pin) for pin in sound_pins]
for button in buttons:
    sound = sound_pins[button.pin.number]
    button.when_pressed = sound.play

pause()
```

See GPIO Music Box for a full resource.

6.1.9 All on when pressed

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

FishDish:

```
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`:

```
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`, `Buzzer`, and `Button` components:

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

button = Button(2)
buzzer = Buzzer(3)
red = LED(4)
amber = LED(5)
green = LED(6)

things = [red, amber, green, buzzer]

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

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

button.when_pressed = things_on
button.when_released = things_off

pause()
```

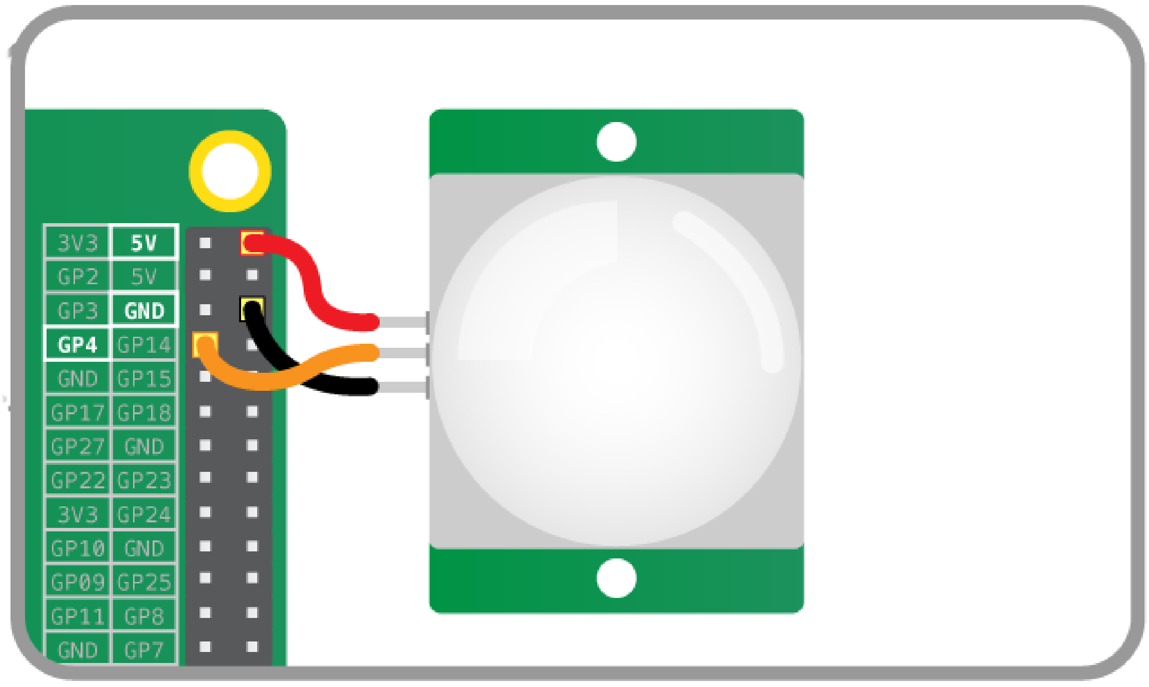
6.1.10 RGB LED

Making colours with an `RGBLED`:

```
from gpiozero import RGBLED
from time import sleep

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

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

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

```
sleep(1)
led.color = (1, 1, 1)  # white
sleep(1)

led.color = (0, 0, 0)  # off
sleep(1)

# slowly increase intensity of blue
for n in range(100):
    led.blue = n/100
    sleep(0.1)
```

6.1.11 Motion sensor



Light an *LED* when a *MotionSensor* detects motion:

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

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

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

pause()
```

6.1.12 Light sensor

Have a *LightSensor* detect light and dark:

```
from gpiozero import LightSensor

sensor = LightSensor(18)

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

Run a function when the light changes:

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

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

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

pause()
```

Or make a `PWMLED` change brightness according to the detected light level:

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

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

led.source = sensor.values

pause()
```

6.1.13 Motors

Spin a `Motor` around forwards and backwards:

```
from gpiozero import Motor
from time import sleep

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

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

6.1.14 Robot

Make a `Robot` drive around in (roughly) a square:

```
from gpiozero import Robot
from time import sleep

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

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

6.1.15 Button controlled robot

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

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

robot = RyanteckRobot()

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()
```

6.1.16 Keyboard controlled robot

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

```
from gpiozero import RyanteckRobot
from evdev import InputDevice, list_devices, ecodes

robot = RyanteckRobot()

devices = [InputDevice(device) for device in list_devices()]
keyboard = devices[0]  # this may vary

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:
        if event.value == 1:  # key down
            keypress_actions[event.code]()
        if event.value == 0:  # key up
            robot.stop()
```

6.1.17 Motion sensor robot

Make a robot drive forward when it detects motion:

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

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

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

pause()
```

Alternatively:

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

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

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

pause()
```

6.1.18 Potentiometer

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

```
from gpiozero import MCP3008

while True:
    with MCP3008(channel=0) as pot:
        print(pot.value)
```

6.1.19 Measure temperature with an ADC

Wire a TMP36 temperature sensor to the first channel of an *MCP3008* 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)
```

6.1.20 Full color LED controlled by 3 potentiometers

Wire up three potentiometers (for red, green and blue) and use each of their values to make up the colour of the LED:

```
from gpiozero import RGBLED, MCP3008

led = RGBLED(red=2, green=3, blue=4)
red_pot = MCP3008(channel=0)
green_pot = MCP3008(channel=1)
```

```
blue_pot = MCP3008(channel=2)

while True:
    led.red = red_pot.value
    led.green = green_pot.value
    led.blue = blue_pot.value
```

Alternatively, the following example is identical, but uses the `source` property rather than a `while` loop:

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

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

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

pause()
```

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

6.2 Notes

6.2.1 Keep your script running

The following script looks like it should turn an LED on:

```
from gpiozero import LED

led = LED(17)
led.on()
```

And it does, if you're using the Python (or IPython or IDLE) shell. However, if you saved this script as a Python file and ran it, it would flash on briefly, then the script would end and it would turn off.

The following file includes an intentional `pause()` to keep the script alive:

```
from gpiozero import LED
from signal import pause

led = LED(17)
led.on()
pause()
```

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

```
from gpiozero import Button
from signal import pause

def hello():
    print("Hello")

button = Button(2)
button.when_pressed = hello
pause()
```

6.2.2 Importing from 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:

```
from gpiozero import Button
```

Now `Button` is available directly in your script:

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

Alternatively, the whole GPIO Zero library can be imported:

```
import gpiozero
```

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

```
button = gpiozero.Button(2)
```

6.3 Input Devices

These input device component interfaces have been provided for simple use of everyday components. Components must be wired up correctly before use in code.

Note: All GPIO pin numbers use Broadcom (BCM) numbering. See the Recipes page for more information.

6.3.1 Button

class `gpiozero.`**`Button`**(*pin*, *pull_up=True*, *bounce_time=None*)

Extends `DigitalInputDevice` and represents a simple push button or switch.

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

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

```
from gpiozero import Button

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

Parameters

- **`pin`** (*`int`*) – The GPIO pin which the button is attached to. See Notes for valid pin numbers.
- **`pull_up`** (*`bool`*) – 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.
- **`bounce_time`** (*`float`*) – If `None` (the default), no software bounce compensation will be performed. Otherwise, this is the length in time (in seconds) that the component will ignore changes in state after an initial change.

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

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

is_pressed
Returns `True` if the device is currently active and `False` otherwise.

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

pull_up
If `True`, the device uses a pull-up resistor to set the GPIO pin "high" by default. Defaults to `False`.

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

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

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

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

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

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

6.3.2 Motion Sensor (PIR)

class gpiozero.**MotionSensor**(*pin*, *queue_len=1*, *sample_rate=10*, *threshold=0.5*, *partial=False*)
Extends *SmoothedInputDevice* and represents a passive infra-red (PIR) motion sensor like the sort found in the CamJam #2 EduKit.

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

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

```python
from gpiozero import MotionSensor

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

Parameters

- **pin** (*int*) – The GPIO pin which the button is attached to. See Notes for valid pin numbers.

- **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. Defaults to 10.
- **threshold** (*float*) – Defaults to 0.5. When the mean of all values in the internal queue rises above this value, the sensor will be considered "active" by the *is_active* property, and all appropriate events will be fired.
- **partial** (*bool*) – When `False` (the default), the object will not return a value for *is_active* until the internal queue has filled with values. Only set this to `True` if you require values immediately after object construction.

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

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

motion_detected
Returns `True` if the device is currently active and `False` otherwise.

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

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

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

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

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

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

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

6.3.3 Light Sensor (LDR)

class gpiozero.**LightSensor**(*pin*, *queue_len=5*, *charge_time_limit=0.01*, *threshold=0.1*, *partial=False*)
Extends *SmoothedInputDevice* 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:

```
from gpiozero import LightSensor

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

Parameters

- **pin** (*int*) – The GPIO pin which the button is attached to. See Notes for valid pin numbers.
- **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 0.01μf 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 mean of all values in the internal queue rises above this value, the area will be considered "light", and all appropriate events will be fired.
- **partial** (*bool*) – When `False` (the default), the object will not return a value for *is_active* until the internal queue has filled with values. Only set this to `True` if you require values immediately after object construction.

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

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

light_detected
Returns `True` if the device is currently active and `False` otherwise.

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

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

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

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

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

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

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

6.3.4 Analog to Digital Converters (ADC)

class `gpiozero.`**`MCP3004`**(*channel=0*, *device=0*, *differential=False*)

The MCP3004 is a 10-bit analog to digital converter with 4 channels (0-3).

`bus`

The SPI bus that the device is connected to. As the Pi only has a single (user accessible) SPI bus, this always returns 0.

`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), and the MCP3301 only has 1 channel.

`device`

The select pin that the device is connected to. The Pi has two select pins so this will be 0 or 1.

`differential`

If `True`, the device is operated in pseudo-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`* in differential mode, channel 0 is read relative to channel 1).

`value`

The current value read from the device, scaled to a value between 0 and 1.

class `gpiozero.`**`MCP3008`**(*channel=0*, *device=0*, *differential=False*)

The MCP3008 is a 10-bit analog to digital converter with 8 channels (0-7).

`bus`

The SPI bus that the device is connected to. As the Pi only has a single (user accessible) SPI bus, this always returns 0.

`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), and the MCP3301 only has 1 channel.

`device`

The select pin that the device is connected to. The Pi has two select pins so this will be 0 or 1.

`differential`

If `True`, the device is operated in pseudo-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`* in differential mode, channel 0 is read relative to channel 1).

`value`

The current value read from the device, scaled to a value between 0 and 1.

class `gpiozero.`**`MCP3204`**(*channel=0*, *device=0*, *differential=False*)

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

`bus`

The SPI bus that the device is connected to. As the Pi only has a single (user accessible) SPI bus, this always returns 0.

`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), and the MCP3301 only has 1 channel.

`device`

The select pin that the device is connected to. The Pi has two select pins so this will be 0 or 1.

`differential`

If `True`, the device is operated in pseudo-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* in differential mode, channel 0 is read relative to channel 1).

value
The current value read from the device, scaled to a value between 0 and 1.

class gpiozero.**MCP3208**(*channel=0*, *device=0*, *differential=False*)
The MCP3208 is a 12-bit analog to digital converter with 8 channels (0-7).

bus
The SPI bus that the device is connected to. As the Pi only has a single (user accessible) SPI bus, this always returns 0.

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), and the MCP3301 only has 1 channel.

device
The select pin that the device is connected to. The Pi has two select pins so this will be 0 or 1.

differential
If `True`, the device is operated in pseudo-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* in differential mode, channel 0 is read relative to channel 1).

value
The current value read from the device, scaled to a value between 0 and 1.

class gpiozero.**MCP3301**(*device=0*)
The MCP3301 is a signed 13-bit analog to digital converter. Please note that the MCP3301 always operates in differential mode between its two channels and the output value is scaled from -1 to +1.

bus
The SPI bus that the device is connected to. As the Pi only has a single (user accessible) SPI bus, this always returns 0.

device
The select pin that the device is connected to. The Pi has two select pins so this will be 0 or 1.

value
The current value read from the device, scaled to a value between 0 and 1.

class gpiozero.**MCP3302**(*channel=0*, *device=0*, *differential=False*)
The MCP3302 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.

bus
The SPI bus that the device is connected to. As the Pi only has a single (user accessible) SPI bus, this always returns 0.

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), and the MCP3301 only has 1 channel.

device
The select pin that the device is connected to. The Pi has two select pins so this will be 0 or 1.

differential
If `True`, the device is operated in pseudo-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* in differential mode, channel 0 is read relative to channel 1).

value
The current value read from the device, scaled to a value between 0 and 1.

class gpiozero.**MCP3304**(*channel=0*, *device=0*, *differential=False*)
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.

bus
The SPI bus that the device is connected to. As the Pi only has a single (user accessible) SPI bus, this always returns 0.

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), and the MCP3301 only has 1 channel.

device
The select pin that the device is connected to. The Pi has two select pins so this will be 0 or 1.

differential
If `True`, the device is operated in pseudo-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* in differential mode, channel 0 is read relative to channel 1).

value
The current value read from the device, scaled to a value between 0 and 1.

6.4 Output Devices

These output device component interfaces have been provided for simple use of everyday components. Components must be wired up correctly before use in code.

Note: All GPIO pin numbers use Broadcom (BCM) numbering. See the Recipes page for more information.

6.4.1 LED

class gpiozero.**LED**(*pin*, *active_high=True*, *initial_value=False*)
Extends *DigitalOutputDevice* and represents a light emitting diode (LED).

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

The following example will light the LED:

```
from gpiozero import LED

led = LED(17)
led.on()
```

Parameters

- **pin** (*int*) – The GPIO pin which the LED is attached to. See Notes for valid pin numbers.
- **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`*) – 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.

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

`off`()
Turns the device off.

`on`()
Turns the device on.

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

`is_lit`
Returns `True` if the device is currently active and `False` otherwise.

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

6.4.2 PWMLED

class `gpiozero.`**`PWMLED`**(*pin*, *active_high=True*, *initial_value=0*, *frequency=100*)
Extends *`PWMOutputDevice`* 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`*) – The GPIO pin which the LED is attached to. See Notes for valid pin numbers.
- **`active_high`** (*`bool`*) – If `True` (the default), the *`on()`* method will set the GPIO to HIGH. If `False`, the *`on()`* method will set the GPIO to LOW (the *`off()`* method always does the opposite).
- **`initial_value`** (*`bool`*) – 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.

`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*) – 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()

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

is_lit

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

pin

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

value

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

6.4.3 RGBLED

class `gpiozero.`**RGBLED**(*red*, *green*, *blue*, *active_high=True*, *initial_value=(0, 0, 0)*)

Extends `CompositeDevice` and represents a full color LED component (composed of red, green, and blue LEDs).

Connect the common cathode (longest leg) to a ground pin; connect each of the other legs (representing the red, green, and blue anodes) to any GPIO pins. You can either use three limiting resistors (one per anode) or a single limiting resistor on the cathode.

The following code will make the LED purple:

```
from gpiozero import RGBLED

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

Parameters

- **red** (*int*) – The GPIO pin that controls the red component of the RGB LED.
- **green** (*int*) – The GPIO pin that controls the green component of the RGB LED.
- **blue** (*int*) – 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** (*bool*) – The initial color for the LED. Defaults to black `(0, 0, 0)`.

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.
- **fade_out_time** (*float*) – Number of seconds to spend fading out. Defaults to 0.
- **on_color** (*tuple*) – The color to use when the LED is "on". Defaults to white.
- **off_color** (*tuple*) – The color to use when the LED is "off". Defaults to black.
- **n** (*int*) – 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)`.

toggle()
Toggle the state of the device. If the device is currently off (`value` is `(0, 0, 0)`), this changes it to "fully" on (`value` is `(1, 1, 1)`). If the device has a specific color, this method inverts the color.

color
Represents the color of the LED as an RGB 3-tuple of `(red, green, blue)` where each value is between 0 and 1.

For example, purple would be `(1, 0, 1)` and yellow would be `(1, 1, 0)`, while orange would be `(1, 0.5, 0)`.

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

6.4.4 Buzzer

class gpiozero.**Buzzer**(*pin*, *active_high=True*, *initial_value=False*)
Extends *DigitalOutputDevice* and represents a digital buzzer component.

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

The following example will sound the buzzer:

```python
from gpiozero import Buzzer

bz = Buzzer(3)
bz.on()
```

Parameters

- **pin** (*int*) – The GPIO pin which the buzzer is attached to. See Notes for valid pin numbers.
- **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*) – 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.

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

off()
Turns the device off.

on()
Turns the device on.

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

is_active
Returns True if the device is currently active and False otherwise.

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

6.4.5 Motor

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

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

The following code will make the motor turn "forwards":

```
from gpiozero import Motor

motor = Motor(17, 18)
motor.forward()
```

Parameters

- **forward** (*int*) – The GPIO pin that the forward input of the motor driver chip is connected to.
- **backward** (*int*) – The GPIO pin that the backward input of the motor driver chip is connected to.

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).

stop()
Stop the motor.

6.5 Boards and Accessories

These additional interfaces are provided to group collections of components together for ease of use, and as examples. They are composites made up of components from the various Input Devices and Output Devices provided by GPIO Zero. See those pages for more information on using components individually.

Note: All GPIO pin numbers use Broadcom (BCM) numbering. See the Recipes page for more information.

6.5.1 LED Board

class gpiozero.**LEDBoard**(**pins*, *pwm=False*)
Extends *CompositeDevice* and represents a generic LED board or collection of LEDs.

The following example turns on all the LEDs on a board containing 5 LEDs attached to GPIO pins 2 through 6:

```
from gpiozero import LEDBoard

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

Parameters

- ***pins** (*int*) – Specify the GPIO pins that the LEDs of the board are attached to. You can designate as many pins as necessary.
- **pwm** (*bool*) – If True, construct *PWMLED* instances for each pin. If False (the default), construct regular *LED* instances. This parameter can only be specified as a keyword parameter.

blink(*on_time=1*, *off_time=1*, *fade_in_time=0*, *fade_out_time=0*, *n=None*, *background=True*)
Make all the LEDs turn on and off repeatedly.

Parameters

- **on_time** (*float*) – Number of seconds on. Defaults to 1 second.
- **off_time** (*float*) – Number of seconds off. Defaults to 1 second.
- **fade_in_time** (*float*) – Number of seconds to spend fading in. Defaults to 0. Must be 0 if pwm was False when the class was constructed (ValueError will be raised if not).
- **fade_out_time** (*float*) – Number of seconds to spend fading out. Defaults to 0. Must be 0 if pwm was False when the class was constructed (ValueError will be raised if not).
- **n** (*int*) – Number of times to blink; None (the default) means forever.

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

close()
Shut down the device and release all associated resources.

off()
Turn all the LEDs off.

on()
Turn all the LEDs on.

toggle()
Toggle all the LEDs. For each LED, if it's on, turn it off; if it's off, turn it on.

leds
A tuple of all the *LED* or *PWMLED* objects contained by the instance.

source
The iterable to use as a source of values for *value*.

value
A tuple containing a value for each LED on the board. This property can also be set to update the state of all LEDs on the board.

values
An infinite iterator of values read from *value*.

6.5.2 LED Bar Graph

class gpiozero.**LEDBarGraph**(**pins*, *initial_value=0*)
Extends *CompositeDevice* 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 turns on all the LEDs on a board containing 5 LEDs attached to GPIO pins 2 through 6:

```
from gpiozero import LEDBarGraph

graph = LEDBarGraph(2, 3, 4, 5, 6)
graph.value = 2/5  # Light the first two LEDs only
graph.value = -2/5 # Light the last two LEDs only
graph.off()
```

As with other output devices, *source* and *values* are supported:

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

graph = LEDBarGraph(2, 3, 4, 5, 6)
pot = MCP3008(channel=0)
graph.source = pot.values
pause()
```

Parameters

- ***pins** (*int*) – Specify the GPIO pins that the LEDs of the bar graph are attached to. You can designate as many pins as necessary.
- **initial_value** (*float*) – The initial *value* of the graph given as a float between -1 and +1. Defaults to 0.0.

close()
Shut down the device and release all associated resources.

off()
Turn all the LEDs off.

on()
Turn all the LEDs on.

toggle()
Toggle all the LEDs. For each LED, if it's on, turn it off; if it's off, turn it on.

leds
A tuple of all the *LED* or *PWMLED* objects contained by the instance.

source
The iterable to use as a source of values for *value*.

value
The value of the LED bar graph. When no LEDs are lit, the value is 0. When all LEDs are lit, the value is 1. Values between 0 and 1 light LEDs linearly from first to last. Values between 0 and -1 light LEDs linearly from last to first.

To light a particular number of LEDs, simply divide that number by the number of LEDs. For example, if your graph contains 3 LEDs, the following will light the first:

```
from gpiozero import LEDBarGraph

graph = LEDBarGraph(12, 16, 19)
graph.value = 1/3
```

Note: Setting value to -1 will light all LEDs. However, querying it subsequently will return 1 as both representations are the same in hardware.

values
An infinite iterator of values read from *value*.

6.5.3 Traffic Lights

class gpiozero.**TrafficLights**(*red=None*, *amber=None*, *green=None*, *pwm=False*)
Extends *LEDBoard* for devices containing red, amber, and green LEDs.

The following example initializes a device connected to GPIO pins 2, 3, and 4, then lights the amber LED attached to GPIO 3:

```
from gpiozero import TrafficLights

traffic = TrafficLights(2, 3, 4)
traffic.amber.on()
```

Parameters

- **red** (*int*) – The GPIO pin that the red LED is attached to.
- **amber** (*int*) – The GPIO pin that the amber LED is attached to.
- **green** (*int*) – The GPIO pin that the green LED is attached to.
- **pwm** (*bool*) – If True, construct *PWMLED* instances to represent each LED. If False (the default), construct regular *LED* instances.

blink(*on_time=1*, *off_time=1*, *fade_in_time=0*, *fade_out_time=0*, *n=None*, *background=True*)
Make all the LEDs turn on and off repeatedly.

Parameters

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

close()
Shut down the device and release all associated resources.

off()
Turn all the LEDs off.

on()
Turn all the LEDs on.

toggle()
Toggle all the LEDs. For each LED, if it's on, turn it off; if it's off, turn it on.

amber
The *LED* or *PWMLED* object representing the red LED.

green
The *LED* or *PWMLED* object representing the green LED.

leds
A tuple of all the *LED* or *PWMLED* objects contained by the instance.

red
The *LED* or *PWMLED* object representing the red LED.

source
The iterable to use as a source of values for *value*.

value
A 3-tuple containing values for the red, amber, and green LEDs. This property can also be set to alter the state of the LEDs.

values
An infinite iterator of values read from *value*.

6.5.4 PiLITEr

class gpiozero.**PiLiter**(*pwm=False*)
Extends *LEDBoard* for the Ciseco Pi-LITEr: a strip of 8 very bright LEDs.

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

```python
from gpiozero import PiLiter

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

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

blink (*on_time=1*, *off_time=1*, *fade_in_time=0*, *fade_out_time=0*, *n=None*, *background=True*)
Make all the LEDs turn on and off repeatedly.

Parameters

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

close ()
Shut down the device and release all associated resources.

off ()
Turn all the LEDs off.

on ()
Turn all the LEDs on.

toggle ()
Toggle all the LEDs. For each LED, if it's on, turn it off; if it's off, turn it on.

leds
A tuple of all the *LED* or *PWMLED* objects contained by the instance.

source
The iterable to use as a source of values for *value*.

value
A tuple containing a value for each LED on the board. This property can also be set to update the state of all LEDs on the board.

values
An infinite iterator of values read from *value*.

6.5.5 PiLITEr Bar Graph

class gpiozero.**PiLiterBarGraph** (*initial_value=0*)
Extends *LEDBarGraph* 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:

```
from gpiozero import PiLiterBarGraph

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

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

close()
Shut down the device and release all associated resources.

off()
Turn all the LEDs off.

on()
Turn all the LEDs on.

toggle()
Toggle all the LEDs. For each LED, if it's on, turn it off; if it's off, turn it on.

leds
A tuple of all the *LED* or *PWMLED* objects contained by the instance.

source
The iterable to use as a source of values for *value*.

value
The value of the LED bar graph. When no LEDs are lit, the value is 0. When all LEDs are lit, the value is 1. Values between 0 and 1 light LEDs linearly from first to last. Values between 0 and -1 light LEDs linearly from last to first.

To light a particular number of LEDs, simply divide that number by the number of LEDs. For example, if your graph contains 3 LEDs, the following will light the first:

```
from gpiozero import LEDBarGraph

graph = LEDBarGraph(12, 16, 19)
graph.value = 1/3
```

Note: Setting value to -1 will light all LEDs. However, querying it subsequently will return 1 as both representations are the same in hardware.

values
An infinite iterator of values read from *value*.

6.5.6 PI-TRAFFIC

class gpiozero.**PiTraffic**
Extends *TrafficLights* 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:

```
from gpiozero import PiTraffic

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

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

blink(*on_time=1*, *off_time=1*, *fade_in_time=0*, *fade_out_time=0*, *n=None*, *background=True*)
Make all the LEDs turn on and off repeatedly.

Parameters

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

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

close()
Shut down the device and release all associated resources.

off()
Turn all the LEDs off.

on()
Turn all the LEDs on.

toggle()
Toggle all the LEDs. For each LED, if it's on, turn it off; if it's off, turn it on.

amber
The *LED* or *PWMLED* object representing the red LED.

green
The *LED* or *PWMLED* object representing the green LED.

leds
A tuple of all the *LED* or *PWMLED* objects contained by the instance.

red
The *LED* or *PWMLED* object representing the red LED.

source
The iterable to use as a source of values for *value*.

value
A 3-tuple containing values for the red, amber, and green LEDs. This property can also be set to alter the state of the LEDs.

values
An infinite iterator of values read from *value*.

6.5.7 TrafficLightsBuzzer

class gpiozero.**TrafficLightsBuzzer**(*lights*, *buzzer*, *button*)
Extends *CompositeDevice* and is a generic class for HATs with traffic lights, a button and a buzzer.

Parameters

- **lights** (`TrafficLights`) – An instance of *TrafficLights* representing the traffic lights of the HAT.
- **buzzer** (`Buzzer`) – An instance of *Buzzer* representing the buzzer on the HAT.
- **button** (`Button`) – An instance of *Button* representing the button on the HAT.

blink(*on_time=1*, *off_time=1*, *n=None*, *background=True*)
Make all the board's components turn on and off repeatedly.

Parameters

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

close()
Shut down the device and release all associated resources.

off()
Turn all the board's components off.

on()
Turn all the board's components on.

toggle()
Toggle all the board's components. For each component, if it's on, turn it off; if it's off, turn it on.

all
A tuple containing objects for all the items on the board (several *LED* objects, a *Buzzer*, and a *Button*).

source
The iterable to use as a source of values for *value*.

value
Returns a named-tuple containing values representing the states of the LEDs, and the buzzer. This property can also be set to a 4-tuple to update the state of all the board's components.

values
An infinite iterator of values read from *value*.

6.5.8 Fish Dish

class gpiozero.**FishDish**(*pwm=False*)
Extends *TrafficLightsBuzzer* 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:

```
from gpiozero import FishDish

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

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

blink(*on_time=1*, *off_time=1*, *n=None*, *background=True*)
Make all the board's components turn on and off repeatedly.

Parameters

- **on_time** (*float*) – Number of seconds on
- **off_time** (*float*) – Number of seconds off
- **n** (*int*) – Number of times to blink; None means forever

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

close()
Shut down the device and release all associated resources.

off()
Turn all the board's components off.

on()
Turn all the board's components on.

toggle()
Toggle all the board's components. For each component, if it's on, turn it off; if it's off, turn it on.

all
A tuple containing objects for all the items on the board (several *LED* objects, a *Buzzer*, and a *Button*).

source
The iterable to use as a source of values for *value*.

value
Returns a named-tuple containing values representing the states of the LEDs, and the buzzer. This property can also be set to a 4-tuple to update the state of all the board's components.

values
An infinite iterator of values read from *value*.

6.5.9 Traffic HAT

class gpiozero.**TrafficHat**(*pwm=False*)
Extends *TrafficLightsBuzzer* for the Ryanteck Traffic HAT: traffic light LEDs, a button and a buzzer.

The Traffic HAT pins are fixed and therefore there's no need to specify them when constructing this class. The following example waits for the button to be pressed on the Traffic HAT, then turns on all the LEDs:

```
from gpiozero import TrafficHat

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

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

blink(*on_time=1*, *off_time=1*, *n=None*, *background=True*)
Make all the board's components turn on and off repeatedly.

Parameters

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

close()
Shut down the device and release all associated resources.

off()
Turn all the board's components off.

on()
Turn all the board's components on.

toggle()
Toggle all the board's components. For each component, if it's on, turn it off; if it's off, turn it on.

all
A tuple containing objects for all the items on the board (several `LED` objects, a `Buzzer`, and a `Button`).

source
The iterable to use as a source of values for *value*.

value
Returns a named-tuple containing values representing the states of the LEDs, and the buzzer. This property can also be set to a 4-tuple to update the state of all the board's components.

values
An infinite iterator of values read from *value*.

6.5.10 Robot

class gpiozero.**Robot**(*left=None*, *right=None*)
Extends `CompositeDevice` to represent a generic dual-motor robot.

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

```
from gpiozero import Robot

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

Parameters

- **left** (`tuple`) – A tuple of two GPIO pins representing the forward and backward inputs of the left motor's controller.
- **right** (`tuple`) – A tuple of two GPIO pins representing the forward and backward inputs of the right motor's controller.

backward(*speed=1*)
Drive the robot backward by running both motors backward.

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

close()
Shut down the device and release all associated resources.

forward(*speed=1*)
Drive the robot forward by running both motors forward.

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

left(*speed=1*)
Make the robot turn left by running the right motor forward and left motor backward.

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

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

right(*speed=1*)
Make the robot turn right by running the left motor forward and right motor backward.

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

stop()
Stop the robot.

left_motor
Returns the *Motor* device representing the robot's left motor.

right_motor
Returns the *Motor* device representing the robot's right motor.

source
The iterable to use as a source of values for *value*.

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

values
An infinite iterator of values read from *value*.

6.5.11 Ryanteck MCB Robot

class gpiozero.**RyanteckRobot**
Extends *Robot* for the Ryanteck MCB robot.

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

```
from gpiozero import RyanteckRobot

robot = RyanteckRobot()
robot.left()
```

backward(*speed=1*)
Drive the robot backward by running both motors backward.

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

close()
Shut down the device and release all associated resources.

forward(*speed=1*)
Drive the robot forward by running both motors forward.

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

left(*speed=1*)
Make the robot turn left by running the right motor forward and left motor backward.

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

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

right(*speed=1*)
Make the robot turn right by running the left motor forward and right motor backward.

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

stop()
Stop the robot.

left_motor
Returns the *Motor* device representing the robot's left motor.

right_motor
Returns the *Motor* device representing the robot's right motor.

source
The iterable to use as a source of values for *value*.

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

values
An infinite iterator of values read from *value*.

6.5.12 CamJam #3 Kit Robot

class gpiozero.**CamJamKitRobot**
Extends *Robot* for the CamJam #3 EduKit robot controller.

The CamJam robot controller pins are fixed and therefore there's no need to specify them when constructing this class. The following example turns the robot left:

```
from gpiozero import CamJamKitRobot

robot = CamJamKitRobot()
robot.left()
```

backward(*speed=1*)
Drive the robot backward by running both motors backward.

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

close()
Shut down the device and release all associated resources.

forward(*speed=1*)
Drive the robot forward by running both motors forward.

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

left(*speed=1*)
Make the robot turn left by running the right motor forward and left motor backward.

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

reverse()
Reverse the robot's current motor directions. If the robot is currently running full speed forward, it

will run full speed backward. If the robot is turning left at half-speed, it will turn right at half-speed. If the robot is currently stopped it will remain stopped.

right (*speed=1*)
Make the robot turn right by running the left motor forward and right motor backward.

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

stop ()
Stop the robot.

left_motor
Returns the *Motor* device representing the robot's left motor.

right_motor
Returns the *Motor* device representing the robot's right motor.

source
The iterable to use as a source of values for *value*.

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

values
An infinite iterator of values read from *value*.

6.6 Generic Devices

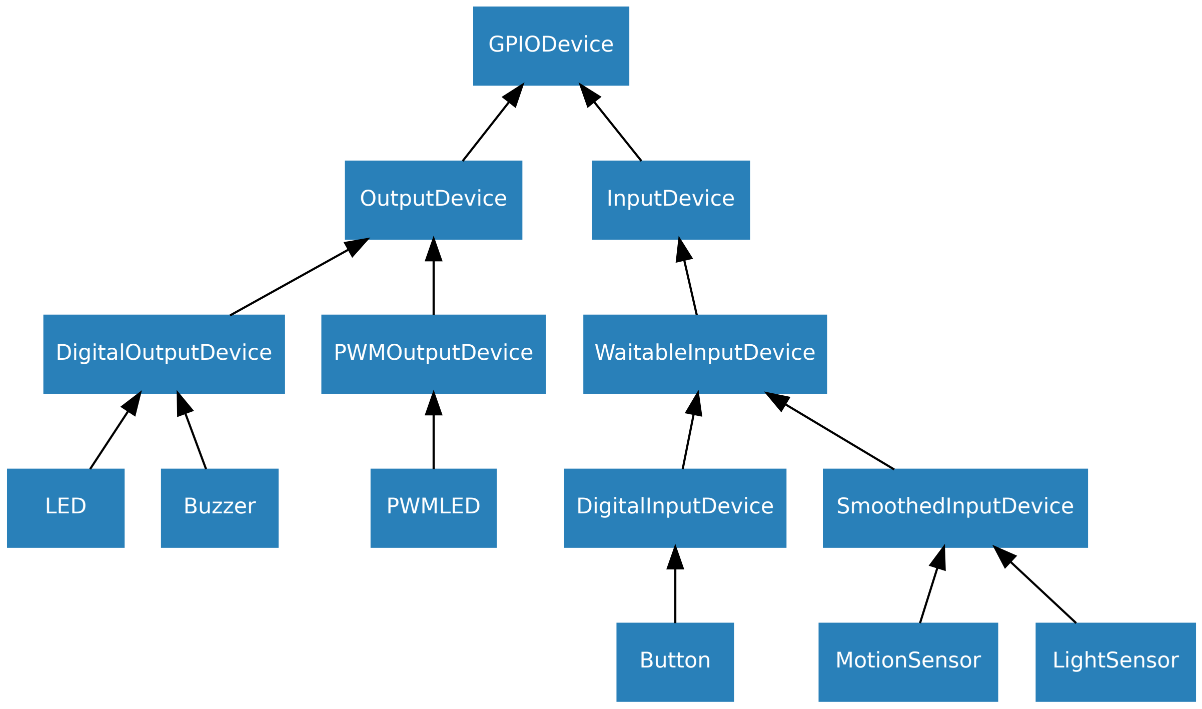
The GPIO Zero class hierarchy is quite extensive. It contains a couple of base classes:

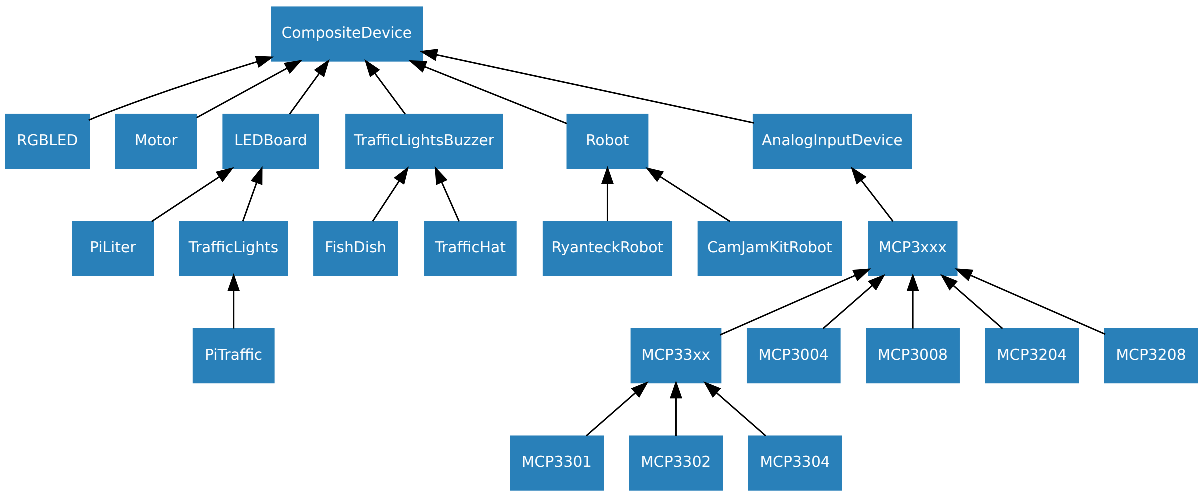
- *GPIODevice* for individual devices that attach to a single GPIO pin
- *CompositeDevice* for devices composed of multiple other devices like HATs

There are also a couple of mixin classes:

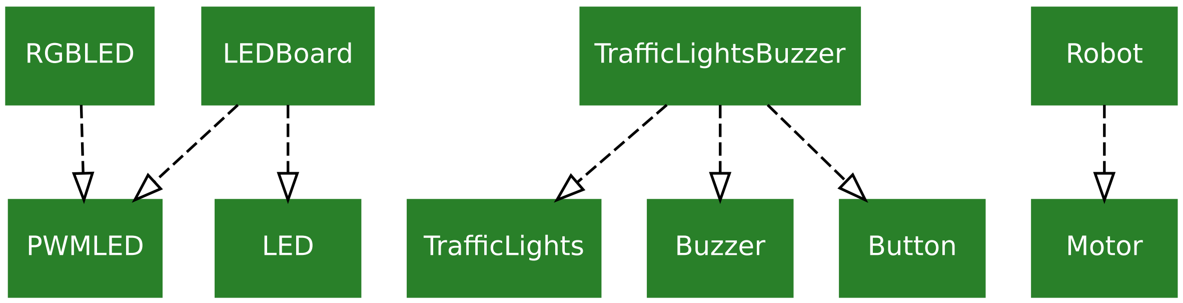
- *ValuesMixin* which defines the `values` properties; there is rarely a need to use this as the base classes mentioned above both include it (so all classes in GPIO Zero include the `values` property)
- *SourceMixin* which defines the `source` property; this is generally included in novel output device classes

The current class hierarchies are displayed below. For brevity, the mixin classes are omitted:





Finally, for composite devices, the following chart shows which devices are composed of which other devices:



6.6.1 Base Classes

class gpiozero.**GPIODevice**(*pin*)

Represents a generic GPIO device.

This is the class at the root of the gpiozero class hierarchy. It handles ensuring that two GPIO devices do not share the same pin, and provides basic services applicable to all devices (specifically the `pin` property, *is_active* property, and the *close* method).

Parameters **pin** (*int*) – The GPIO pin (in BCM numbering) that the device is connected to. If this is `None` a `GPIODeviceError` will be raised.

close()

Shut down the device and release all associated resources.

This method is primarily intended for interactive use at the command line. It disables the device and releases its pin for use by another device.

You can attempt to do this simply by deleting an object, but unless you've cleaned up all references to the object this may not work (even if you've cleaned up all references, there's still no guarantee the garbage collector will actually delete the object at that point). By contrast, the close method provides a means of ensuring that the object is shut down.

For example, if you have a breadboard with a buzzer connected to pin 16, but then wish to attach an LED instead:

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

GPIODevice descendents can also be used as context managers using the `with` statement. For example:

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

is_active

Returns `True` if the device is currently active and `False` otherwise.

pin

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

value

Returns `True` if the device is currently active and `False` otherwise.

values

An infinite iterator of values read from *value*.

class gpiozero.**CompositeDevice**

Represents a device composed of multiple GPIO devices like simple HATs, H-bridge motor controllers, robots composed of multiple motors, etc.

close()

Shut down the device and release all associated resources.

closed

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

values

An infinite iterator of values read from *value*.

6.6.2 Input Devices

class `gpiozero.`**`InputDevice`**(*pin*, *pull_up=False*)

Represents a generic GPIO input device.

This class extends `GPIODevice` to add facilities common to GPIO input devices. The constructor adds the optional *pull_up* parameter to specify how the pin should be pulled by the internal resistors. The `is_active` property is adjusted accordingly so that `True` still means active regardless of the `pull_up` setting.

Parameters

- **`pin`** (*`int`*) – The GPIO pin (in Broadcom numbering) that the device is connected to. If this is `None` a `GPIODeviceError` will be raised.
- **`pull_up`** (*`bool`*) – If `True`, the pin will be pulled high with an internal resistor. If `False` (the default), the pin will be pulled low.

`pull_up`

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

class `gpiozero.`**`WaitableInputDevice`**(*pin=None*, *pull_up=False*)

Represents a generic input device with distinct waitable states.

This class extends `InputDevice` with methods for waiting on the device's status (`wait_for_active()` and `wait_for_inactive()`), and properties that hold functions to be called when the device changes state (`when_activated()` and `when_deactivated()`). These are aliased appropriately in various subclasses.

Note that this class provides no means of actually firing its events; it's effectively an abstract base class.

`wait_for_active`(*timeout=None*)

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

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

`when_activated`

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

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

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

`when_deactivated`

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

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

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

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

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

This class extends `WaitableInputDevice` 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 **bouncetime** (*float*) – 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.

class `gpiozero.`**`SmoothedInputDevice`**(*pin=None*, *pull_up=False*, *threshold=0.5*, *queue_len=5*, *sample_wait=0.0*, *partial=False*)

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

This class extends *WaitableInputDevice* with a queue which is filled by a background thread which continually polls the state of the underlying device. The mean of the values in the queue is compared to a threshold which is used to determine the state of the *is_active* property.

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

Parameters

- **threshold** (*float*) – 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* property) will block until the queue has filled. If `True`, a value will be returned immediately, but be aware that this value is likely to fluctuate excessively.

close()

Shut down the device and release all associated resources.

This method is primarily intended for interactive use at the command line. It disables the device and releases its pin for use by another device.

You can attempt to do this simply by deleting an object, but unless you've cleaned up all references to the object this may not work (even if you've cleaned up all references, there's still no guarantee the garbage collector will actually delete the object at that point). By contrast, the close method provides a means of ensuring that the object is shut down.

For example, if you have a breadboard with a buzzer connected to pin 16, but then wish to attach an LED instead:

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

GPIODevice descendents can also be used as context managers using the `with` statement. For example:

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

is_active

Returns `True` if the device is currently active and `False` otherwise.

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

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

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

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

class `gpiozero.`**AnalogInputDevice**(*device=0*, *bits=None*)
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*, *MCP3008*, *MCP3204*, and *MCP3208*.

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

```
from gpiozero import MCP3008

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

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

```
from gpiozero import MCP3008, PWMLED

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

close()
Shut down the device and release all associated resources.

bits
The bit-resolution of the device/channel.

bus
The SPI bus that the device is connected to. As the Pi only has a single (user accessible) SPI bus, this always returns 0.

device
The select pin that the device is connected to. The Pi has two select pins so this will be 0 or 1.

raw_value
The raw value as read from the device.

value
The current value read from the device, scaled to a value between 0 and 1.

6.6.3 Output Devices

class `gpiozero.`**OutputDevice**(*pin*, *active_high=True*, *initial_value=False*)
Represents a generic GPIO output device.

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

Parameters

- **pin** (*int*) – The GPIO pin (in BCM numbering) that the device is connected to. If this is `None` a `GPIODeviceError` will be raised.
- **active_high** (*bool*) – If `True` (the default), the *on()* method will set the GPIO to HIGH. If `False`, the *on()* method will set the GPIO to LOW (the *off()* method always does the opposite).
- **initial_value** (*bool*) – 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.

off()

Turns the device off.

on()

Turns the device on.

class gpiozero.**PWMOutputDevice**(*pin*, *active_high=True*, *initial_value=0*, *frequency=100*)

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

Parameters

- **pin** (*int*) – The GPIO pin which the device is attached to. See Notes for valid pin numbers.
- **active_high** (*bool*) – If `True` (the default), the *on()* method will set the GPIO to HIGH. If `False`, the *on()* method will set the GPIO to LOW (the *off()* method always does the opposite).
- **initial_value** (*bool*) – 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.

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*) – 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).

close()

Shut down the device and release all associated resources.

This method is primarily intended for interactive use at the command line. It disables the device and releases its pin for use by another device.

You can attempt to do this simply by deleting an object, but unless you've cleaned up all references to the object this may not work (even if you've cleaned up all references, there's still no guarantee the garbage collector will actually delete the object at that point). By contrast, the close method provides a means of ensuring that the object is shut down.

For example, if you have a breadboard with a buzzer connected to pin 16, but then wish to attach an LED instead:

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

`GPIODevice` descendents can also be used as context managers using the `with` statement. For example:

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

off()
Turns the device off.

on()
Turns the device on.

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

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

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

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

class gpiozero.**DigitalOutputDevice**(*pin*, *active_high=True*, *initial_value=False*)
Represents a generic output device with typical on/off behaviour.

This class extends `OutputDevice` with a `toggle()` method to switch the device between its on and off states, and a `blink()` method which uses an optional background thread to handle toggling the device state without further interaction.

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

Parameters

- **on_time** (*float*) – Number of seconds on. Defaults to 1 second.
- **off_time** (*float*) – Number of seconds off. Defaults to 1 second.
- **n** (*int*) – 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).

close()
Shut down the device and release all associated resources.

This method is primarily intended for interactive use at the command line. It disables the device and releases its pin for use by another device.

You can attempt to do this simply by deleting an object, but unless you've cleaned up all references to the object this may not work (even if you've cleaned up all references, there's still no guarantee the garbage collector will actually delete the object at that point). By contrast, the close method provides a means of ensuring that the object is shut down.

For example, if you have a breadboard with a buzzer connected to pin 16, but then wish to attach an LED instead:

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

GPIODevice descendents can also be used as context managers using the `with` statement. For example:

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

off()
Turns the device off.

on()
Turns the device on.

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

6.6.4 Mixin Classes

class gpiozero.**ValuesMixin**(...)

values
An infinite iterator of values read from *value*.

class gpiozero.**SourceMixin**(...)

source
The iterable to use as a source of values for *value*.

6.7 Pins

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 GPIO pin number. However, what the library actually expects is a `Pin` implementation. If it finds a simple integer number instead, it uses one of the following classes to provide the `Pin` implementation (classes are listed in favoured order):

1. `gpiozero.pins.rpigpio.RPiGPIOPin`
2. `gpiozero.pins.rpio.RPIOPin`
3. `gpiozero.pins.native.NativePin`

You can change the default pin implementation by over-writing the `DefaultPin` global in devices like so:

```
from gpiozero.pins.native import NativePin
import gpiozero.devices
# Force the default pin implementation to be NativePin
gpiozero.devices.DefaultPin = NativePin

from gpiozero import LED

# This will now use NativePin instead of RPiGPIOPin
led = LED(16)
```

In future, this separation should allow the library to utilize pins that are part of IO extender chips. For example:

```
from gpiozero import IOExtender, LED

ext = IOExtender()
led = LED(ext.pins[0])
led.on()
```

Warning: While the devices API is now considered stable and won't change in backwards incompatible ways, the pins API is *not* yet considered stable. It is potentially subject to change in future versions. We welcome any comments from testers!

6.7.1 Abstract Pin

class `gpiozero.`**`Pin`**

Abstract base class representing a GPIO pin or a pin from an IO extender.

Descendents should override property getters and setters to accurately represent the capabilities of pins. The following functions *must* be overridden:

- `_get_function()`
- `_get_state()`

The following functions *may* be overridden if applicable:

- `close()`
- `_set_function()`
- `_set_state()`
- `_get_frequency()`
- `_set_frequency()`
- `_get_pull()`
- `_set_pull()`
- `_get_bounce()`
- `_set_bounce()`
- `_get_edges()`

- `_set_edges()`
- `_get_when_changed()`
- `_set_when_changed()`
- *`output_with_state()`*
- *`input_with_pull()`*

> **Warning:** Descendents must ensure that pin instances representing the same physical hardware are identical, right down to object identity. The framework relies on this to correctly clean up resources at interpreter shutdown.

`close``()`

Cleans up the resources allocated to the pin. After this method is called, this *`Pin`* instance may no longer be used to query or control the pin's state.

`input_with_pull``(`*pull*`)`

Sets the pin's function to "input" and specifies an initial pull-up for the pin. By default this is equivalent to performing:

```
pin.function = 'input'
pin.pull = pull
```

However, descendents may override this order to provide the smallest possible delay between configuring the pin for input and pulling the pin up/down (which can be important for avoiding "blips" in some configurations).

`output_with_state``(`*state*`)`

Sets the pin's function to "output" and specifies an initial state for the pin. By default this is equivalent to performing:

```
pin.function = 'output'
pin.state = state
```

However, descendents may override this in order to provide the smallest possible delay between configuring the pin for output and specifying an initial value (which can be important for avoiding "blips" in active-low configurations).

`bounce`

The amount of bounce detection (elimination) currently in use by edge detection, measured in seconds. If bounce detection is not currently in use, this is `None`.

If the pin does not support edge detection, attempts to set this property will raise `PinEdgeDetectUnsupported`. If the pin supports edge detection, the class must implement bounce detection, even if only in software.

`edges`

The edge that will trigger execution of the function or bound method assigned to *`when_changed`*. This can be one of the strings "both" (the default), "rising", "falling", or "none".

If the pin does not support edge detection, attempts to set this property will raise `PinEdgeDetectUnsupported`.

`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` will be raised when attempting to set this to a value other than `None`.

`function`

The function of the pin. This property is a string indicating the current function or purpose of the pin.

Typically this is the string "input" or "output". However, in some circumstances it can be other strings indicating non-GPIO related functionality.

With certain pin types (e.g. GPIO pins), this attribute can be changed to configure the function of a pin. If an invalid function is specified, for this attribute, `PinInvalidFunction` will be raised. If this pin is fixed function and an attempt is made to set this attribute, `PinFixedFunction` will be raised.

`pull`
The pull-up state of the pin represented as a string. This is typically one of the strings "up", "down", or "floating" but additional values may be supported by the underlying hardware.

If the pin does not support changing pull-up state (for example because of a fixed pull-up resistor), attempts to set this property will raise `PinFixedPull`. If the specified value is not supported by the underlying hardware, `PinInvalidPull` is raised.

`state`
The state of the pin. This is 0 for low, and 1 for high. As a low level view of the pin, no swapping is performed in the case of pull ups (see *`pull`* for more information).

If PWM is currently active (when *`frequency`* is not `None`), this represents the PWM duty cycle as a value between 0.0 and 1.0.

If a pin is currently configured for input, and an attempt is made to set this attribute, `PinSetInput` will be raised. If an invalid value is specified for this attribute, `PinInvalidState` will be raised.

`when_changed`
A function or bound method to be called when the pin's state changes (more specifically when the edge specified by *`edges`* is detected on the pin). The function or bound method must take no parameters.

If the pin does not support edge detection, attempts to set this property will raise `PinEdgeDetectUnsupported`.

6.7.2 RPiGPIOPin

class `gpiozero.pins.rpigpio.`**`RPiGPIOPin`**
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).

6.7.3 RPIOPin

class `gpiozero.pins.rpio.`**`RPIOPin`**
Uses the RPIO library to interface to the Pi's GPIO pins. This is the default pin implementation if the RPi.GPIO library is not installed, but RPIO is. Supports all features including PWM (hardware via DMA).

Note: Please note that at the time of writing, RPIO is only compatible with Pi 1's; the Raspberry Pi 2 Model B is *not* supported. Also note that root access is required so scripts must typically be run with `sudo`.

6.7.4 NativePin

class `gpiozero.pins.native.`**`NativePin`**
Uses a built-in pure Python implementation to interface to the Pi's GPIO pins. This is the default pin implementation if no third-party libraries are discovered.

Warning: This implementation does *not* currently support PWM. Attempting to use any class which requests PWM will raise an exception. This implementation is also experimental; we make no guarantees it will not eat your Pi for breakfast!

6.8 Changelog

6.8.1 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 *LEDBarGraph* class (many thanks to Martin O'Hanlon!) (#126, #176)
- New *Pin* 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()* method which works properly even when background is set to `False` (#94, #161)
- New *RGBLED.blink()* method which implements (rudimentary) color fading too! (#135, #174)
- New `initial_value` attribute on *OutputDevice* ensures consistent behaviour on construction (#118)
- New `active_high` attribute on *PWMOutputDevice* and *RGBLED* allows use of common anode devices (#143, #154)
- Loads of new ADC chips supported (many thanks to GitHub user pcopa!) (#150)

6.8.2 Release 1.0.0 (2015-11-16)

- Debian packaging added (#44)
- *PWMLED* class added (#58)
- `TemperatureSensor` removed pending further work (#93)
- *Buzzer.beep()* alias method added (#75)
- *Motor* PWM devices exposed, and *Robot* motor devices exposed (#107)

6.8.3 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* constructor (#79)
- Added wrappers for LED groups on add-on boards (#81)

6.8.4 Release 0.8.0 (2015-10-16)

Third public beta

- Added generic *AnalogInputDevice* class along with specific classes for the *MCP3008* and *MCP3004* (#41)
- Fixed *DigitalOutputDevice.blink()* (#57)

6.8.5 Release 0.7.0 (2015-10-09)

Second public beta

6.8.6 Release 0.6.0 (2015-09-28)

First public beta

6.8.7 Release 0.5.0 (2015-09-24)

6.8.8 Release 0.4.0 (2015-09-23)

6.8.9 Release 0.3.0 (2015-09-22)

6.8.10 Release 0.2.0 (2015-09-21)

Initial release

6.9 License

Index