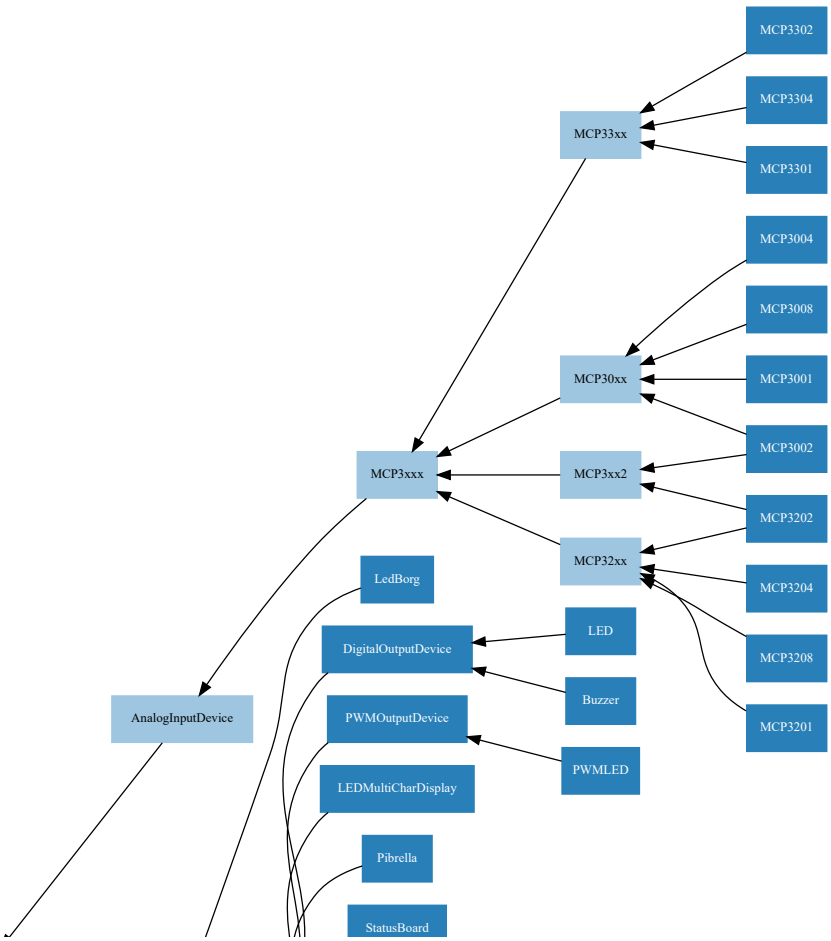
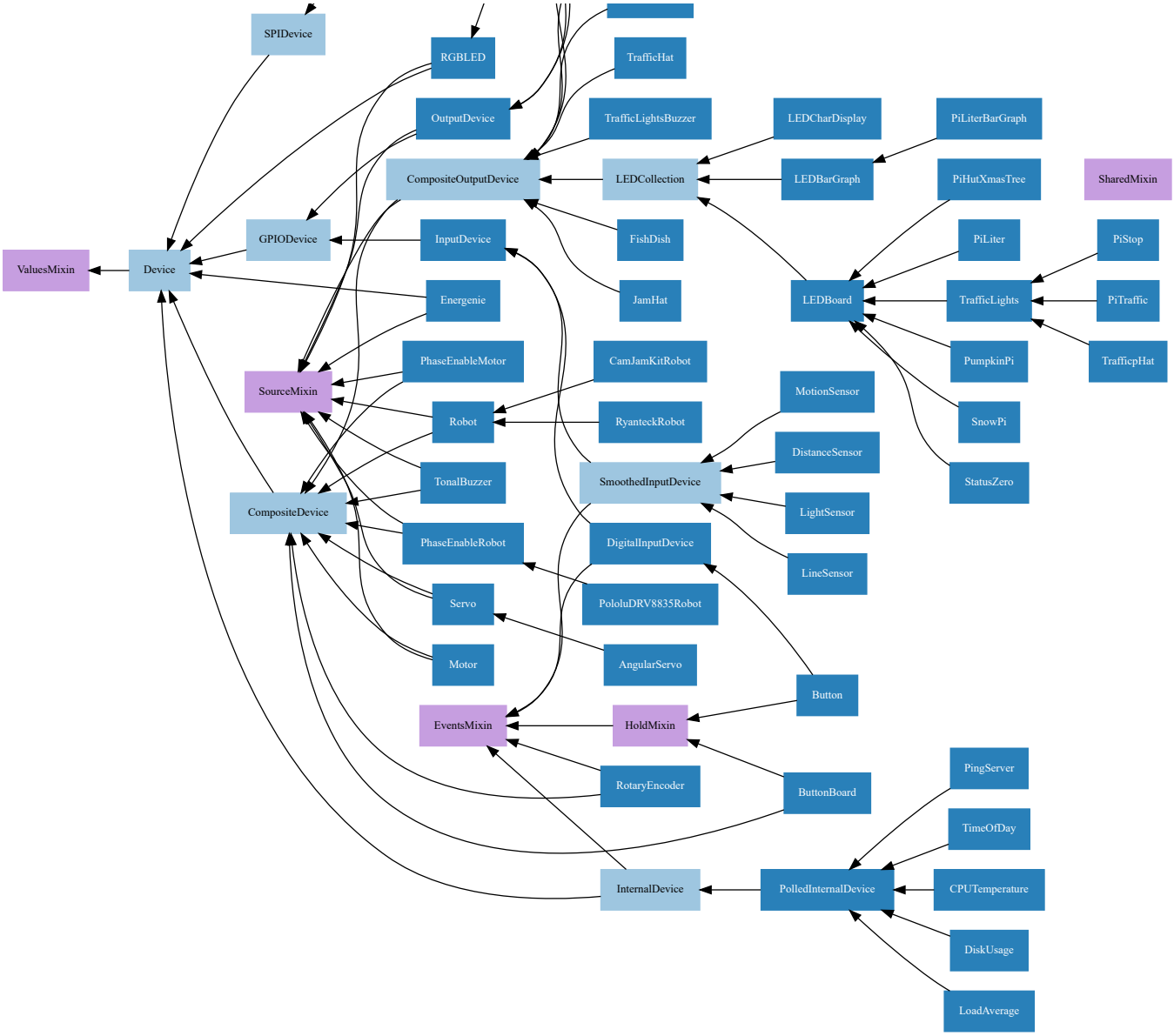
18. API - Generic Classes

The GPIO Zero class hierarchy is quite extensive. It contains several base classes (most of which are documented in their corresponding chapters):

- `Device` is the root of the hierarchy, implementing base functionality like `close()` and context manager handlers.
- `GPIODevice` represents individual devices that attach to a single GPIO pin
- `SPIDevice` represents devices that communicate over an SPI interface (implemented as four GPIO pins)
- `InternalDevice` represents devices that are entirely internal to the Pi (usually operating system related services)
- `CompositeDevice` represents devices composed of multiple other devices like HATs

There are also several mixin classes for adding important functionality at numerous points in the hierarchy, which is illustrated below (mixin classes are represented in purple, while abstract classes are shaded lighter):





18.1. Device

class **gpiozero.Device**(**, pin_factory=None*) [source]

Represents a single device of any type; GPIO-based, SPI-based, I2C-based, etc. This is the base class of the device hierarchy. It defines the basic services applicable to all devices (specifically the `is_active` property, the `value` property, and the `close()` method).

> **pin_factory**
>
> This attribute exists at both a class level (representing the default pin factory used to construct devices when no *pin_factory* parameter is specified), and at an instance level (representing the pin factory that the device was constructed with).
>
> The pin factory provides various facilities to the device including allocating pins, providing low level interfaces (e.g. SPI), and clock facilities (querying and calculating elapsed times).

> **close**()

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

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

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

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

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

`Device` 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()
...
```

`closed`

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

`is_active`

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

`value`

Returns a value representing the device's state. Frequently, this is a boolean value, or a number between 0 and 1 but some devices use larger ranges (e.g. -1 to +1) and composite devices usually use tuples to return the states of all their subordinate components.

18.2. ValuesMixin

class **gpiozero.ValuesMixin(...)** [source]

Adds a `values` property to the class which returns an infinite generator of readings from the `value` property. There is rarely a need to use this mixin directly as all base classes in GPIO Zero include it.

Note

Use this mixin *first* in the parent class list.

values

An infinite iterator of values read from `value`.

18.3. SourceMixin

class **gpiozero.SourceMixin(...)** [source]

Adds a `source` property to the class which, given an iterable or a `ValuesMixin` descendent, sets `value` to each member of that iterable until it is exhausted. This mixin is generally included in novel output devices to allow their state to be driven from another device.

Note

Use this mixin *first* in the parent class list.

source

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

source_delay

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

18.4. SharedMixin

class **gpiozero.SharedMixin**(*...*) [source]

This mixin marks a class as "shared". In this case, the meta-class (GPIOMeta) will use `_shared_key()` to convert the constructor arguments to an immutable key, and will check whether any existing instances match that key. If they do, they will be returned by the constructor instead of a new instance. An internal reference counter is used to determine how many times an instance has been "constructed" in this way.

When `close()` is called, an internal reference counter will be decremented and the instance will only close when it reaches zero.

classmethod **_shared_key**(**args*, ***kwargs*) [source]

This is called with the constructor arguments to generate a unique key (which must be storable in a `dict` and, thus, immutable and hashable) representing the instance that can be shared. This must be overridden by descendents.

The default simply assumes all positional arguments are immutable and returns this as the key but this is almost never the "right" thing to do and almost all descendents should override this method.

18.5. EventsMixin

class **gpiozero.EventsMixin**(*...*) [source]

Adds edge-detected `when_activated()` and `when_deactivated()` events to a device based on changes to the `is_active` property common to all devices. Also adds `wait_for_active()` and `wait_for_inactive()` methods for level-waiting.

Note

Note that this mixin provides no means of actually firing its events; call `_fire_events()` in sub-classes when device state changes to trigger the events. This should also be called once at the end of initialization to set initial states.

wait_for_active(*timeout=None*) [source]

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

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

wait_for_inactive(*timeout=None*) [source]

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

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

active_time

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

inactive_time

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

when_activated

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

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

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

when_deactivated

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

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

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

18.6. HoldMixin

class **gpiozero.HoldMixin**(*...*) [source]

Extends `EventsMixin` to add the `when_held` event and the machinery to fire that event repeatedly (when `hold_repeat` is `True`) at internals defined by `hold_time`.

held_time

The length of time (in seconds) that the device has been held for. This is counted from the first execution of the `when_held` event rather than when the device activated, in contrast to `active_time`. If the device is not currently held, this is `None`.

`hold_repeat`

If `True`, `when_held` will be executed repeatedly with `hold_time` seconds between each invocation.

`hold_time`

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

`is_held`

When `True`, the device has been active for at least `hold_time` seconds.

`when_held`

The function to run when the device has remained active for `hold_time` seconds.

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

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