
python-ev3dev Documentation

Release 1.0.0.post5

Ralph Hempel et al

Jul 28, 2018

Contents

1	Getting Started	3
2	Usage Examples	5
2.1	Required: Import the library	5
2.2	Controlling the LEDs with a touch sensor	5
2.3	Running a motor	6
2.4	Using text-to-speech	6
3	Writing Python Programs for Ev3dev	7
4	User Resources	9
5	Upgrading this Library	11
6	Developer Resources	13
7	Python 2.x and Python 3.x Compatibility	15
7.1	API reference	15
7.1.1	Motor classes	16
7.1.2	Sensor classes	24
7.1.3	Other classes	29
7.2	Working with ev3dev remotely using RPyC	39
7.3	Frequently-Asked Questions	41
7.3.1	My script works when launched as <code>python3 script.py</code> but exits immediately or throws an error when launched from Brickman or as <code>./script.py</code>	41
8	Indices and tables	43

A Python3 library implementing an interface for `ev3dev` devices, letting you control motors, sensors, hardware buttons, LCD displays and more from Python code.

If you haven't written code in Python before, you'll need to learn the language before you can use this library.

CHAPTER 1

Getting Started

This library runs on [ev3dev](#). Before continuing, make sure that you have set up your EV3 or other ev3dev device as explained in the [ev3dev Getting Started guide](#). Make sure that you have a kernel version that includes `-10-ev3dev` or higher (a larger number). You can check the kernel version by selecting “About” in Brickman and scrolling down to the “kernel version”. If you don’t have a compatible version, [upgrade the kernel before continuing](#). Also note that if the ev3dev image you downloaded was created before September 2016, you probably don’t have the most recent version of this library installed: see [Upgrading this Library](#) to upgrade it.

Once you have booted ev3dev and [connected to your EV3 \(or Raspberry Pi / BeagleBone\) via SSH](#), you should be ready to start using ev3dev with Python: this library is included out-of-the-box. If you want to go through some basic usage examples, check out the [Usage Examples](#) section to try out motors, sensors and LEDs. Then look at [Writing Python Programs for Ev3dev](#) to see how you can save your Python code to a file.

Make sure that you look at the [User Resources](#) section as well for links to documentation and larger examples.

CHAPTER 2

Usage Examples

To run these minimal examples, run the Python3 interpreter from the terminal using the `python3` command:

```
$ python3
Python 3.4.2 (default, Oct  8 2014, 14:47:30)
[GCC 4.9.1] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

The `>>>` characters are the default prompt for Python. In the examples below, we have removed these characters so it's easier to cut and paste the code into your session.

2.1 Required: Import the library

If you are using an EV3 brick (which is the case for most users), add the following to the top of your file:

```
import ev3dev.ev3 as ev3
```

If you are using a BrickPi, use this line:

```
import ev3dev.brickpi as ev3
```

2.2 Controlling the LEDs with a touch sensor

This code will turn the left LED red whenever the touch sensor is pressed, and back to green when it's released. Plug a touch sensor into any sensor port and then paste in this code - you'll need to hit `Enter` after pasting to complete the loop and start the program. Hit `Ctrl-C` to exit the loop.

```
ts = ev3.TouchSensor()
while True:
    ev3.Leds.set_color(ev3.Leds.LEFT, (ev3.Leds.GREEN, ev3.Leds.RED)[ts.value()])
```

2.3 Running a motor

Now plug a motor into the A port and paste this code into the Python prompt. This little program will run the motor at 500 ticks per second, which on the EV3 “large” motors equates to around 1.4 rotations per second, for three seconds (3000 milliseconds).

```
m = ev3.LargeMotor('outA')
m.run_timed(time_sp=3000, speed_sp=500)
```

The units for `speed_sp` that you see above are in “tacho ticks” per second. On the large EV3 motor, these equate to one tick per degree, so this is 500 degrees per second.

2.4 Using text-to-speech

If you want to make your robot speak, you can use the *Sound.speak* method:

```
ev3.Sound.speak('Welcome to the E V 3 dev project!').wait()
```

To quit the Python REPL, just type `exit()` or press Ctrl-D.

Make sure to check out the [User Resources](#) section for more detailed information on these features and many others.

Writing Python Programs for Ev3dev

Every Python program should have a few basic parts. Use this template to get started:

```
#!/usr/bin/env python3
from ev3dev.ev3 import *

# TODO: Add code here
```

The first two lines should be included in every Python program you write for ev3dev. The first allows you to run this program from Brickman, while the second imports this library.

When saving Python files, it is best to use the `.py` extension, e.g. `my-file.py`. To be able to run your Python code, **your program must be executable**. To mark a program as executable run `chmod +x my-file.py`. You can then run `my-file.py` via the Brickman File Browser or you can run it from the command line via `$./my-file.py`

CHAPTER 4

User Resources

Library Documentation Class documentation for this library can be found on our [Read the Docs page](#). You can always go there to get information on how you can use this library's functionality.

ev3python.com One of our community members, @ndward, has put together a great website with detailed guides on using this library which are targeted at beginners. If you are just getting started with programming, we highly recommend that you check it out at [ev3python.com](#)!

Frequently-Asked Questions Experiencing an odd error or unsure of how to do something that seems simple? Check our [FAQ](#) to see if there's an existing answer.

ev3dev.org [ev3dev.org](#) is a great resource for finding guides and tutorials on using ev3dev, straight from the maintainers.

Support If you are having trouble using this library, please open an issue at our [Issues tracker](#) so that we can help you. When opening an issue, make sure to include as much information as possible about what you are trying to do and what you have tried. The issue template is in place to guide you through this process.

Demo Robot Laurens Valk of [robot-square](#) has been kind enough to allow us to reference his excellent [EXPLOR3R](#) robot. Consider building the [EXPLOR3R](#) and running the demo programs referenced below to get familiar with what Python programs using this binding look like.

Demo Code There are [demo programs](#) that you can run to get acquainted with this language binding. The programs are designed to work with the [EXPLOR3R](#) robot.

CHAPTER 5

Upgrading this Library

You can upgrade this library from the command line as follows. Make sure to type the password (the default is `maker`) when prompted.

```
sudo apt-get update
sudo apt-get install --only-upgrade python3-ev3dev
```

Developer Resources

Python Package Index The Python language has a [package repository](#) where you can find libraries that others have written, including the [latest version of this package](#).

The ev3dev Binding Specification Like all of the language bindings for ev3dev supported hardware, the Python binding follows the minimal API that must be provided per [this document](#).

The ev3dev-lang Project on GitHub The [source repository for the generic API](#) and the scripts to automatically generate the binding. Only developers of the [ev3dev-lang-python](#) binding would normally need to access this information.

Python 2.x and Python 3.x Compatibility

Some versions of the `ev3dev` distribution come with both `Python 2.x` and `Python 3.x` installed but this library is compatible only with Python 3.

As of the 2016-10-17 `ev3dev` image, the version of this library which is included runs on Python 3 and this is the only version that will be supported from here forward.

Contents

7.1 API reference

Each class in `ev3dev` module inherits from the base `Device` class.

class `ev3dev.core.Device` (*class_name*, *name_pattern*=`'*'`, *name_exact*=`False`, ***kwargs*)
The `ev3dev` device base class

`ev3dev.core.list_device_names` (*class_path*, *name_pattern*, ***kwargs*)
This is a generator function that lists names of all devices matching the provided parameters.

Parameters:

class_path: class path of the device, a subdirectory of `/sys/class`. For example, `‘/sys/class/tachomotor’`.

name_pattern: pattern that device name should match. For example, `‘sensor*’` or `‘motor*’`. Default value: `‘*’`.

keyword arguments: used for matching the corresponding device attributes. For example, `address=‘outA’`, or `driver_name=[‘lego-ev3-us’, ‘lego-nxt-us’]`. When argument value is a list, then a match against any entry of the list is enough.

`ev3dev.core.list_devices` (*class_name*, *name_pattern*, ***kwargs*)

This is a generator function that takes same arguments as `Device` class and enumerates all devices present in the system that match the provided arguments.

Parameters:

class_name: class name of the device, a subdirectory of `/sys/class`. For example, 'tacho-motor'.

name_pattern: pattern that device name should match. For example, 'sensor*' or 'motor*'. Default value: '*'.

keyword arguments: used for matching the corresponding device attributes. For example, `address='outA'`, or `driver_name=['lego-ev3-us', 'lego-nxt-us']`. When argument value is a list, then a match against any entry of the list is enough.

```
ev3dev.core.list_motors(name_pattern='*', **kwargs)
```

This is a generator function that enumerates all tacho motors that match the provided arguments.

Parameters:

name_pattern: pattern that device name should match. For example, 'motor*'. Default value: '*'.

keyword arguments: used for matching the corresponding device attributes. For example, `driver_name='lego-ev3-l-motor'`, or `address=['outB', 'outC']`. When argument value is a list, then a match against any entry of the list is enough.

```
ev3dev.core.list_sensors(name_pattern='sensor*', **kwargs)
```

This is a generator function that enumerates all sensors that match the provided arguments.

Parameters:

name_pattern: pattern that device name should match. For example, 'sensor*'. Default value: '*'.

keyword arguments: used for matching the corresponding device attributes. For example, `driver_name='lego-ev3-touch'`, or `address=['in1', 'in3']`. When argument value is a list, then a match against any entry of the list is enough.

Contents:

7.1.1 Motor classes

Tacho motor

```
class ev3dev.core.Motor(address=None, name_pattern='*', name_exact=False, **kwargs)
```

The motor class provides a uniform interface for using motors with positional and directional feedback such as the EV3 and NXT motors. This feedback allows for precise control of the motors. This is the most common type of motor, so we just call it *motor*.

The way to configure a motor is to set the '_sp' attributes when calling a command or before. Only in 'run_direct' mode attribute changes are processed immediately, in the other modes they only take place when a new command is issued.

```
COMMAND_RESET = 'reset'
```

Reset all of the motor parameter attributes to their default value. This will also have the effect of stopping the motor.

```
COMMAND_RUN_DIRECT = 'run-direct'
```

Run the motor at the duty cycle specified by *duty_cycle_sp*. Unlike other run commands, changing *duty_cycle_sp* while running *will* take effect immediately.

```
COMMAND_RUN_FOREVER = 'run-forever'
```

Run the motor until another command is sent.

```
COMMAND_RUN_TIMED = 'run-timed'
```

Run the motor for the amount of time specified in *time_sp* and then stop the motor using the action specified by *stop_action*.

COMMAND_RUN_TO_ABS_POS = 'run-to-abs-pos'

Run to an absolute position specified by *position_sp* and then stop using the action specified in *stop_action*.

COMMAND_RUN_TO_REL_POS = 'run-to-rel-pos'

Run to a position relative to the current *position* value. The new position will be current *position* + *position_sp*. When the new position is reached, the motor will stop using the action specified by *stop_action*.

COMMAND_STOP = 'stop'

Stop any of the run commands before they are complete using the action specified by *stop_action*.

ENCODER_POLARITY_INVERSED = 'inversed'

Sets the inversed polarity of the rotary encoder.

ENCODER_POLARITY_NORMAL = 'normal'

Sets the normal polarity of the rotary encoder.

POLARITY_INVERSED = 'inversed'

With *inversed* polarity, a positive duty cycle will cause the motor to rotate counter-clockwise.

POLARITY_NORMAL = 'normal'

With *normal* polarity, a positive duty cycle will cause the motor to rotate clockwise.

STATE_HOLDING = 'holding'

The motor is not turning, but rather attempting to hold a fixed position.

STATE_OVERLOADED = 'overloaded'

The motor is turning, but cannot reach its *speed_sp*.

STATE_RAMPING = 'ramping'

The motor is ramping up or down and has not yet reached a constant output level.

STATE_RUNNING = 'running'

Power is being sent to the motor.

STATE_STALLED = 'stalled'

The motor is not turning when it should be.

STOP_ACTION_BRAKE = 'brake'

Power will be removed from the motor and a passive electrical load will be placed on the motor. This is usually done by shorting the motor terminals together. This load will absorb the energy from the rotation of the motors and cause the motor to stop more quickly than coasting.

STOP_ACTION_COAST = 'coast'

Power will be removed from the motor and it will freely coast to a stop.

STOP_ACTION_HOLD = 'hold'

Does not remove power from the motor. Instead it actively try to hold the motor at the current position. If an external force tries to turn the motor, the motor will *push back* to maintain its position.

address

Returns the name of the port that this motor is connected to.

command

Sends a command to the motor controller. See *commands* for a list of possible values.

commands

Returns a list of commands that are supported by the motor controller. Possible values are *run-forever*, *run-to-abs-pos*, *run-to-rel-pos*, *run-timed*, *run-direct*, *stop* and *reset*. Not all commands may be supported.

- *run-forever* will cause the motor to run until another command is sent.
- *run-to-abs-pos* will run to an absolute position specified by *position_sp* and then stop using the action specified in *stop_action*.

- *run-to-rel-pos* will run to a position relative to the current *position* value. The new position will be current *position* + *position_sp*. When the new position is reached, the motor will stop using the action specified by *stop_action*.
- *run-timed* will run the motor for the amount of time specified in *time_sp* and then stop the motor using the action specified by *stop_action*.
- *run-direct* will run the motor at the duty cycle specified by *duty_cycle_sp*. Unlike other run commands, changing *duty_cycle_sp* while running *will* take effect immediately.
- *stop* will stop any of the run commands before they are complete using the action specified by *stop_action*.
- *reset* will reset all of the motor parameter attributes to their default value. This will also have the effect of stopping the motor.

count_per_m

Returns the number of tacho counts in one meter of travel of the motor. Tacho counts are used by the position and speed attributes, so you can use this value to convert from distance to tacho counts. (linear motors only)

count_per_rot

Returns the number of tacho counts in one rotation of the motor. Tacho counts are used by the position and speed attributes, so you can use this value to convert rotations or degrees to tacho counts. (rotation motors only)

driver_name

Returns the name of the driver that provides this tacho motor device.

duty_cycle

Returns the current duty cycle of the motor. Units are percent. Values are -100 to 100.

duty_cycle_sp

Writing sets the duty cycle setpoint. Reading returns the current value. Units are in percent. Valid values are -100 to 100. A negative value causes the motor to rotate in reverse.

full_travel_count

Returns the number of tacho counts in the full travel of the motor. When combined with the *count_per_m* attribute, you can use this value to calculate the maximum travel distance of the motor. (linear motors only)

is_holding

The motor is not turning, but rather attempting to hold a fixed position.

is_overloaded

The motor is turning, but cannot reach its *speed_sp*.

is_ramping

The motor is ramping up or down and has not yet reached a constant output level.

is_running

Power is being sent to the motor.

is_stalled

The motor is not turning when it should be.

max_speed

Returns the maximum value that is accepted by the *speed_sp* attribute. This may be slightly different than the maximum speed that a particular motor can reach - it's the maximum theoretical speed.

polarity

Sets the polarity of the motor. With *normal* polarity, a positive duty cycle will cause the motor to rotate

clockwise. With *inversed* polarity, a positive duty cycle will cause the motor to rotate counter-clockwise. Valid values are *normal* and *inversed*.

position

Returns the current position of the motor in pulses of the rotary encoder. When the motor rotates clockwise, the position will increase. Likewise, rotating counter-clockwise causes the position to decrease. Writing will set the position to that value.

position_d

The derivative constant for the position PID.

position_i

The integral constant for the position PID.

position_p

The proportional constant for the position PID.

position_sp

Writing specifies the target position for the *run-to-abs-pos* and *run-to-rel-pos* commands. Reading returns the current value. Units are in tacho counts. You can use the value returned by *counts_per_rot* to convert tacho counts to/from rotations or degrees.

ramp_down_sp

Writing sets the ramp down setpoint. Reading returns the current value. Units are in milliseconds and must be positive. When set to a non-zero value, the motor speed will decrease from 0 to 100% of *max_speed* over the span of this setpoint. The actual ramp time is the ratio of the difference between the *speed_sp* and the current *speed* and *max_speed* multiplied by *ramp_down_sp*.

ramp_up_sp

Writing sets the ramp up setpoint. Reading returns the current value. Units are in milliseconds and must be positive. When set to a non-zero value, the motor speed will increase from 0 to 100% of *max_speed* over the span of this setpoint. The actual ramp time is the ratio of the difference between the *speed_sp* and the current *speed* and *max_speed* multiplied by *ramp_up_sp*.

reset (***kwargs*)

Reset all of the motor parameter attributes to their default value. This will also have the effect of stopping the motor.

run_direct (***kwargs*)

Run the motor at the duty cycle specified by *duty_cycle_sp*. Unlike other run commands, changing *duty_cycle_sp* while running *will* take effect immediately.

run_forever (***kwargs*)

Run the motor until another command is sent.

run_timed (***kwargs*)

Run the motor for the amount of time specified in *time_sp* and then stop the motor using the action specified by *stop_action*.

run_to_abs_pos (***kwargs*)

Run to an absolute position specified by *position_sp* and then stop using the action specified in *stop_action*.

run_to_rel_pos (***kwargs*)

Run to a position relative to the current *position* value. The new position will be current *position* + *position_sp*. When the new position is reached, the motor will stop using the action specified by *stop_action*.

speed

Returns the current motor speed in tacho counts per second. Note, this is not necessarily degrees (although it is for LEGO motors). Use the *count_per_rot* attribute to convert this value to RPM or deg/sec.

speed_d

The derivative constant for the speed regulation PID.

speed_i

The integral constant for the speed regulation PID.

speed_p

The proportional constant for the speed regulation PID.

speed_sp

Writing sets the target speed in tacho counts per second used for all *run-** commands except *run-direct*. Reading returns the current value. A negative value causes the motor to rotate in reverse with the exception of *run-to-*-pos* commands where the sign is ignored. Use the *count_per_rot* attribute to convert RPM or deg/sec to tacho counts per second. Use the *count_per_m* attribute to convert m/s to tacho counts per second.

state

Reading returns a list of state flags. Possible flags are *running*, *ramping*, *holding*, *overloaded* and *stalled*.

stop (***kwargs*)

Stop any of the run commands before they are complete using the action specified by *stop_action*.

stop_action

Reading returns the current stop action. Writing sets the stop action. The value determines the motors behavior when *command* is set to *stop*. Also, it determines the motors behavior when a run command completes. See *stop_actions* for a list of possible values.

stop_actions

Returns a list of stop actions supported by the motor controller. Possible values are *coast*, *brake* and *hold*. *coast* means that power will be removed from the motor and it will freely coast to a stop. *brake* means that power will be removed from the motor and a passive electrical load will be placed on the motor. This is usually done by shorting the motor terminals together. This load will absorb the energy from the rotation of the motors and cause the motor to stop more quickly than coasting. *hold* does not remove power from the motor. Instead it actively tries to hold the motor at the current position. If an external force tries to turn the motor, the motor will 'push back' to maintain its position.

time_sp

Writing specifies the amount of time the motor will run when using the *run-timed* command. Reading returns the current value. Units are in milliseconds.

wait (*cond*, *timeout=None*)

Blocks until *cond*(*self.state*) is True. The condition is checked when there is an I/O event related to the *state* attribute. Exits early when *timeout* (in milliseconds) is reached.

Returns True if the condition is met, and False if the timeout is reached.

wait_until (*s*, *timeout=None*)

Blocks until *s* is in *self.state*. The condition is checked when there is an I/O event related to the *state* attribute. Exits early when *timeout* (in milliseconds) is reached.

Returns True if the condition is met, and False if the timeout is reached.

Example:

```
m.wait_until('stalled')
```

wait_until_not_moving (*timeout=None*)

Blocks until *running* is not in *self.state* or *stalled* is in *self.state*. The condition is checked when there is an I/O event related to the *state* attribute. Exits early when *timeout* (in milliseconds) is reached.

Returns True if the condition is met, and False if the timeout is reached.

Example:

```
m.wait_until_not_moving()
```

wait_while (*s*, *timeout=None*)

Blocks until *s* is not in *self.state*. The condition is checked when there is an I/O event related to the *state* attribute. Exits early when *timeout* (in milliseconds) is reached.

Returns True if the condition is met, and False if the timeout is reached.

Example:

```
m.wait_while('running')
```

Large EV3 Motor

class `ev3dev.core.LargeMotor` (*address=None*, *name_pattern='*'*, *name_exact=False*, ***kwargs*)

Bases: `ev3dev.core.Motor`

EV3/NXT large servo motor

Medium EV3 Motor

class `ev3dev.core.MediumMotor` (*address=None*, *name_pattern='*'*, *name_exact=False*, ***kwargs*)

Bases: `ev3dev.core.Motor`

EV3 medium servo motor

DC Motor

class `ev3dev.core.DcMotor` (*address=None*, *name_pattern='motor*'*, *name_exact=False*, ***kwargs*)

The DC motor class provides a uniform interface for using regular DC motors with no fancy controls or feedback. This includes LEGO MINDSTORMS RCX motors and LEGO Power Functions motors.

COMMAND_RUN_DIRECT = 'run-direct'

Run the motor at the duty cycle specified by *duty_cycle_sp*. Unlike other run commands, changing *duty_cycle_sp* while running *will* take effect immediately.

COMMAND_RUN_FOREVER = 'run-forever'

Run the motor until another command is sent.

COMMAND_RUN_TIMED = 'run-timed'

Run the motor for the amount of time specified in *time_sp* and then stop the motor using the action specified by *stop_action*.

COMMAND_STOP = 'stop'

Stop any of the run commands before they are complete using the action specified by *stop_action*.

POLARITY_INVERSED = 'inversed'

With *inversed* polarity, a positive duty cycle will cause the motor to rotate counter-clockwise.

POLARITY_NORMAL = 'normal'

With *normal* polarity, a positive duty cycle will cause the motor to rotate clockwise.

STOP_ACTION_BRAKE = 'brake'

Power will be removed from the motor and a passive electrical load will be placed on the motor. This is usually done by shorting the motor terminals together. This load will absorb the energy from the rotation of the motors and cause the motor to stop more quickly than coasting.

STOP_ACTION_COAST = 'coast'

Power will be removed from the motor and it will freely coast to a stop.

address

Returns the name of the port that this motor is connected to.

command

Sets the command for the motor. Possible values are *run-forever*, *run-timed* and *stop*. Not all commands may be supported, so be sure to check the contents of the *commands* attribute.

commands

Returns a list of commands supported by the motor controller.

driver_name

Returns the name of the motor driver that loaded this device. See the list of [supported devices] for a list of drivers.

duty_cycle

Shows the current duty cycle of the PWM signal sent to the motor. Values are -100 to 100 (-100% to 100%).

duty_cycle_sp

Writing sets the duty cycle setpoint of the PWM signal sent to the motor. Valid values are -100 to 100 (-100% to 100%). Reading returns the current setpoint.

polarity

Sets the polarity of the motor. Valid values are *normal* and *inversed*.

ramp_down_sp

Sets the time in milliseconds that it take the motor to ramp down from 100% to 0%. Valid values are 0 to 10000 (10 seconds). Default is 0.

ramp_up_sp

Sets the time in milliseconds that it take the motor to up ramp from 0% to 100%. Valid values are 0 to 10000 (10 seconds). Default is 0.

run_direct (**kwargs)

Run the motor at the duty cycle specified by *duty_cycle_sp*. Unlike other run commands, changing *duty_cycle_sp* while running *will* take effect immediately.

run_forever (**kwargs)

Run the motor until another command is sent.

run_timed (**kwargs)

Run the motor for the amount of time specified in *time_sp* and then stop the motor using the action specified by *stop_action*.

state

Gets a list of flags indicating the motor status. Possible flags are *running* and *ramping*. *running* indicates that the motor is powered. *ramping* indicates that the motor has not yet reached the *duty_cycle_sp*.

stop (**kwargs)

Stop any of the run commands before they are complete using the action specified by *stop_action*.

stop_action

Sets the stop action that will be used when the motor stops. Read *stop_actions* to get the list of valid values.

stop_actions

Gets a list of stop actions. Valid values are *coast* and *brake*.

time_sp

Writing specifies the amount of time the motor will run when using the *run-timed* command. Reading returns the current value. Units are in milliseconds.

Servo Motor

```
class ev3dev.core.ServoMotor (address=None, name_pattern='motor*', name_exact=False,
                               **kwargs)
```

The servo motor class provides a uniform interface for using hobby type servo motors.

COMMAND_FLOAT = 'float'

Remove power from the motor.

COMMAND_RUN = 'run'

Drive servo to the position set in the *position_sp* attribute.

POLARITY_INVERSED = 'inversed'

With *inversed* polarity, a positive duty cycle will cause the motor to rotate counter-clockwise.

POLARITY_NORMAL = 'normal'

With *normal* polarity, a positive duty cycle will cause the motor to rotate clockwise.

address

Returns the name of the port that this motor is connected to.

command

Sets the command for the servo. Valid values are *run* and *float*. Setting to *run* will cause the servo to be driven to the *position_sp* set in the *position_sp* attribute. Setting to *float* will remove power from the motor.

driver_name

Returns the name of the motor driver that loaded this device. See the list of [supported devices] for a list of drivers.

float (**kwargs)

Remove power from the motor.

max_pulse_sp

Used to set the pulse size in milliseconds for the signal that tells the servo to drive to the maximum (clockwise) *position_sp*. Default value is 2400. Valid values are 2300 to 2700. You must write to the *position_sp* attribute for changes to this attribute to take effect.

mid_pulse_sp

Used to set the pulse size in milliseconds for the signal that tells the servo to drive to the mid *position_sp*. Default value is 1500. Valid values are 1300 to 1700. For example, on a 180 degree servo, this would be 90 degrees. On continuous rotation servo, this is the 'neutral' *position_sp* where the motor does not turn. You must write to the *position_sp* attribute for changes to this attribute to take effect.

min_pulse_sp

Used to set the pulse size in milliseconds for the signal that tells the servo to drive to the minimum (counter-clockwise) *position_sp*. Default value is 600. Valid values are 300 to 700. You must write to the *position_sp* attribute for changes to this attribute to take effect.

polarity

Sets the polarity of the servo. Valid values are *normal* and *inversed*. Setting the value to *inversed* will cause the *position_sp* value to be inversed. i.e -100 will correspond to *max_pulse_sp*, and 100 will correspond to *min_pulse_sp*.

position_sp

Reading returns the current `position_sp` of the servo. Writing instructs the servo to move to the specified `position_sp`. Units are percent. Valid values are -100 to 100 (-100% to 100%) where -100 corresponds to `min_pulse_sp`, 0 corresponds to `mid_pulse_sp` and 100 corresponds to `max_pulse_sp`.

rate_sp

Sets the `rate_sp` at which the servo travels from 0 to 100.0% (half of the full range of the servo). Units are in milliseconds. Example: Setting the `rate_sp` to 1000 means that it will take a 180 degree servo 2 second to move from 0 to 180 degrees. Note: Some servo controllers may not support this in which case reading and writing will fail with `-EOPNOTSUPP`. In continuous rotation servos, this value will affect the `rate_sp` at which the speed ramps up or down.

run (**kwargs)

Drive servo to the position set in the `position_sp` attribute.

state

Returns a list of flags indicating the state of the servo. Possible values are: * `running`: Indicates that the motor is powered.

7.1.2 Sensor classes

Sensor

This is the base class all the other sensor classes are derived from.

class `ev3dev.core.Sensor` (`address=None`, `name_pattern='sensor*'`, `name_exact=False`, `**kwargs`)

The sensor class provides a uniform interface for using most of the sensors available for the EV3. The various underlying device drivers will create a `lego-sensor` device for interacting with the sensors.

Sensors are primarily controlled by setting the `mode` and monitored by reading the `value<N>` attributes. Values can be converted to floating point if needed by `value<N> / 10.0 ^ decimals`.

Since the name of the `sensor<N>` device node does not correspond to the port that a sensor is plugged in to, you must look at the `address` attribute if you need to know which port a sensor is plugged in to. However, if you don't have more than one sensor of each type, you can just look for a matching `driver_name`. Then it will not matter which port a sensor is plugged in to - your program will still work.

address

Returns the name of the port that the sensor is connected to, e.g. `ev3:in1`. I2C sensors also include the I2C address (decimal), e.g. `ev3:in1:i2c8`.

bin_data (fmt=None)

Returns the unscaled raw values in the `value<N>` attributes as raw byte array. Use `bin_data_format`, `num_values` and the individual sensor documentation to determine how to interpret the data.

Use `fmt` to unpack the raw bytes into a struct.

Example:

```
>>> from ev3dev import *
>>> ir = InfraredSensor()
>>> ir.value()
28
>>> ir.bin_data('<b')
(28,)
```

bin_data_format

Returns the format of the values in `bin_data` for the current mode. Possible values are:

- *u8*: Unsigned 8-bit integer (byte)
- *s8*: Signed 8-bit integer (sbyte)
- *u16*: Unsigned 16-bit integer (ushort)
- *s16*: Signed 16-bit integer (short)
- *s16_be*: Signed 16-bit integer, big endian
- *s32*: Signed 32-bit integer (int)
- *float*: IEEE 754 32-bit floating point (float)

command

Sends a command to the sensor.

commands

Returns a list of the valid commands for the sensor. Returns -EOPNOTSUPP if no commands are supported.

decimals

Returns the number of decimal places for the values in the *value<N>* attributes of the current mode.

driver_name

Returns the name of the sensor device/driver. See the list of [supported sensors] for a complete list of drivers.

mode

Returns the current mode. Writing one of the values returned by *modes* sets the sensor to that mode.

modes

Returns a list of the valid modes for the sensor.

num_values

Returns the number of *value<N>* attributes that will return a valid value for the current mode.

units

Returns the units of the measured value for the current mode. May return empty string

value (*n=0*)

Returns the value or values measured by the sensor. Check *num_values* to see how many values there are. Values with *N* >= *num_values* will return an error. The values are fixed point numbers, so check decimals to see if you need to divide to get the actual value.

Special sensor classes

The classes derive from *Sensor* and provide helper functions specific to the corresponding sensor type. Each of the functions makes sure the sensor is in the required mode and then returns the specified value.

Touch Sensor

```
class ev3dev.core.TouchSensor (address=None, name_pattern='sensor*', name_exact=False,
                                **kwargs)
```

Bases: *ev3dev.core.Sensor*

Touch Sensor

```
MODE_TOUCH = 'TOUCH'
```

Button state

is_pressed

A boolean indicating whether the current touch sensor is being pressed.

Color Sensor

```
class ev3dev.core.ColorSensor(address=None, name_pattern='sensor*', name_exact=False,
                               **kwargs)
```

Bases: `ev3dev.core.Sensor`

LEGO EV3 color sensor.

COLOR_BLACK = 1

Black color.

COLOR_BLUE = 2

Blue color.

COLOR_BROWN = 7

Brown color.

COLOR_GREEN = 3

Green color.

COLOR_NOCOLOR = 0

No color.

COLOR_RED = 5

Red color.

COLOR_WHITE = 6

White color.

COLOR_YELLOW = 4

Yellow color.

MODE_COL_AMBIENT = 'COL-AMBIENT'

Ambient light. Red LEDs off.

MODE_COL_COLOR = 'COL-COLOR'

Color. All LEDs rapidly cycling, appears white.

MODE_COL_REFLECT = 'COL-REFLECT'

Reflected light. Red LED on.

MODE_REF_RAW = 'REF-RAW'

Raw reflected. Red LED on

MODE_RGB_RAW = 'RGB-RAW'

Raw Color Components. All LEDs rapidly cycling, appears white.

ambient_light_intensity

Ambient light intensity. Light on sensor is dimly lit blue.

blue

Blue component of the detected color, in the range 0-1020.

color

Color detected by the sensor, categorized by overall value.

- 0: No color
- 1: Black

- 2: Blue
- 3: Green
- 4: Yellow
- 5: Red
- 6: White
- 7: Brown

green

Green component of the detected color, in the range 0-1020.

raw

Red, green, and blue components of the detected color, in the range 0-1020.

red

Red component of the detected color, in the range 0-1020.

reflected_light_intensity

Reflected light intensity as a percentage. Light on sensor is red.

Ultrasonic Sensor

```
class ev3dev.core.UltrasonicSensor (address=None,          name_pattern='sensor*',
                                   name_exact=False, **kwargs)
```

Bases: *ev3dev.core.Sensor*

LEGO EV3 ultrasonic sensor.

MODE_US_DIST_CM = 'US-DIST-CM'

Continuous measurement in centimeters.

MODE_US_DIST_IN = 'US-DIST-IN'

Continuous measurement in inches.

MODE_US_LISTEN = 'US-LISTEN'

Listen.

MODE_US_SI_CM = 'US-SI-CM'

Single measurement in centimeters.

MODE_US_SI_IN = 'US-SI-IN'

Single measurement in inches.

distance_centimeters

Measurement of the distance detected by the sensor, in centimeters.

distance_inches

Measurement of the distance detected by the sensor, in inches.

other_sensor_present

Value indicating whether another ultrasonic sensor could be heard nearby.

Gyro Sensor

```
class ev3dev.core.GyroSensor (address=None,  name_pattern='sensor*',  name_exact=False,
                              **kwargs)
```

Bases: *ev3dev.core.Sensor*

LEGO EV3 gyro sensor.

MODE_GYRO_ANG = 'GYRO-ANG'
Angle

MODE_GYRO_CAL = 'GYRO-CAL'
Calibration ???

MODE_GYRO_FAS = 'GYRO-FAS'
Raw sensor value

MODE_GYRO_G_A = 'GYRO-G&A'
Angle and rotational speed

MODE_GYRO_RATE = 'GYRO-RATE'
Rotational speed

angle
The number of degrees that the sensor has been rotated since it was put into this mode.

rate
The rate at which the sensor is rotating, in degrees/second.

rate_and_angle
Angle (degrees) and Rotational Speed (degrees/second).

Infrared Sensor

```
class ev3dev.core.InfraredSensor (address=None, name_pattern='sensor*', name_exact=False,  
                                  **kwargs)
```

Bases: [ev3dev.core.Sensor](#)

LEGO EV3 infrared sensor.

MODE_IR_CAL = 'IR-CAL'
Calibration ???

MODE_IR_PROX = 'IR-PROX'
Proximity

MODE_IR_REMOTE = 'IR-REMOTE'
IR Remote Control

MODE_IR_REM_A = 'IR-REM-A'
IR Remote Control. State of the buttons is coded in binary

MODE_IR_SEEK = 'IR-SEEK'
IR Seeker

proximity
A measurement of the distance between the sensor and the remote, as a percentage. 100% is approximately 70cm/27in.

Sound Sensor

```
class ev3dev.core.SoundSensor (address=None, name_pattern='sensor*', name_exact=False,  
                               **kwargs)
```

Bases: [ev3dev.core.Sensor](#)

LEGO NXT Sound Sensor

MODE_DB = 'DB'

Sound pressure level. Flat weighting

MODE_DBA = 'DBA'

Sound pressure level. A weighting

sound_pressure

A measurement of the measured sound pressure level, as a percent. Uses a flat weighting.

sound_pressure_low

A measurement of the measured sound pressure level, as a percent. Uses A-weighting, which focuses on levels up to 55 dB.

Light Sensor

```
class ev3dev.core.LightSensor(address=None, name_pattern='sensor*', name_exact=False,  
                               **kwargs)
```

Bases: *ev3dev.core.Sensor*

LEGO NXT Light Sensor

MODE_AMBIENT = 'AMBIENT'

Ambient light. LED off

MODE_REFLECT = 'REFLECT'

Reflected light. LED on

ambient_light_intensity

A measurement of the ambient light intensity, as a percentage.

reflected_light_intensity

A measurement of the reflected light intensity, as a percentage.

7.1.3 Other classes

Remote Control

```
class ev3dev.core.RemoteControl(sensor=None, channel=1)  
    EV3 Remote Controller
```

Event handlers

These will be called when state of the corresponding button is changed:

on_red_up

on_red_down

on_blue_up

on_blue_down

on_beacon

Member functions and properties

any()
Checks if any button is pressed.

beacon
Checks if *beacon* button is pressed.

blue_down
Checks if *blue_down* button is pressed.

blue_up
Checks if *blue_up* button is pressed.

buttons_pressed
Returns list of currently pressed buttons.

check_buttons (*buttons=[]*)
Check if currently pressed buttons exactly match the given list.

on_beacon = None
Handles Beacon events.

on_blue_down = None
Handles Blue Down events.

on_blue_up = None
Handles Blue Up events.

static on_change (*changed_buttons*)
This handler is called by *process()* whenever state of any button has changed since last *process()* call. *changed_buttons* is a list of tuples of changed button names and their states.

on_red_down = None
Handles Red Down events.

on_red_up = None
Handles Red Up events.

process()
Check for currently pressed buttons. If the new state differs from the old state, call the appropriate button event handlers.

red_down
Checks if *red_down* button is pressed.

red_up
Checks if *red_up* button is pressed.

Beacon Seeker

class `ev3dev.core.BeaconSeeker` (*sensor=None, channel=1*)
Seeks EV3 Remote Controller in beacon mode.

distance
Returns distance (0, 100) to the beacon on the given channel. Returns -128 when beacon is not found.

heading
Returns heading (-25, 25) to the beacon on the given channel.

heading_and_distance

Returns heading and distance to the beacon on the given channel as a tuple.

Button

class ev3dev.ev3.Button

EV3 Buttons

Event handlers

These will be called when state of the corresponding button is changed:

on_up

on_down

on_left

on_right

on_enter

on_backspace

Member functions and properties

any()

Checks if any button is pressed.

backspace

Check if 'backspace' button is pressed.

buttons_pressed

Returns list of names of pressed buttons.

check_buttons (*buttons=[]*)

Check if currently pressed buttons exactly match the given list.

down

Check if 'down' button is pressed.

enter

Check if 'enter' button is pressed.

left

Check if 'left' button is pressed.

static on_backspace (*state*)

This handler is called by *process()* whenever state of 'backspace' button has changed since last *process()* call. *state* parameter is the new state of the button.

static on_change (*changed_buttons*)

This handler is called by *process()* whenever state of any button has changed since last *process()* call. *changed_buttons* is a list of tuples of changed button names and their states.

static on_down (*state*)

This handler is called by *process()* whenever state of 'down' button has changed since last *process()* call. *state* parameter is the new state of the button.

static on_enter (*state*)

This handler is called by *process()* whenever state of ‘enter’ button has changed since last *process()* call. *state* parameter is the new state of the button.

static on_left (*state*)

This handler is called by *process()* whenever state of ‘left’ button has changed since last *process()* call. *state* parameter is the new state of the button.

static on_right (*state*)

This handler is called by *process()* whenever state of ‘right’ button has changed since last *process()* call. *state* parameter is the new state of the button.

static on_up (*state*)

This handler is called by *process()* whenever state of ‘up’ button has changed since last *process()* call. *state* parameter is the new state of the button.

process ()

Check for currently pressed buttons. If the new state differs from the old state, call the appropriate button event handlers.

right

Check if ‘right’ button is pressed.

up

Check if ‘up’ button is pressed.

Leds

class `ev3dev.core.Led` (*address=None, name_pattern='*', name_exact=False, **kwargs*)

Any device controlled by the generic LED driver. See <https://www.kernel.org/doc/Documentation/leds/leds-class.txt> for more details.

brightness

Sets the brightness level. Possible values are from 0 to *max_brightness*.

brightness_pct

Returns led brightness as a fraction of *max_brightness*

delay_off

The *timer* trigger will periodically change the LED brightness between 0 and the current brightness setting. The *off* time can be specified via *delay_off* attribute in milliseconds.

delay_on

The *timer* trigger will periodically change the LED brightness between 0 and the current brightness setting. The *on* time can be specified via *delay_on* attribute in milliseconds.

max_brightness

Returns the maximum allowable brightness value.

trigger

Sets the led trigger. A trigger is a kernel based source of led events. Triggers can either be simple or complex. A simple trigger isn’t configurable and is designed to slot into existing subsystems with minimal additional code. Examples are the *ide-disk* and *nand-disk* triggers.

Complex triggers whilst available to all LEDs have LED specific parameters and work on a per LED basis. The *timer* trigger is an example. The *timer* trigger will periodically change the LED brightness between 0 and the current brightness setting. The *on* and *off* time can be specified via *delay_{on,off}* attributes in milliseconds. You can change the brightness value of a LED independently of the timer trigger. However, if you set the brightness value to 0 it will also disable the *timer* trigger.

triggers

Returns a list of available triggers.

class `ev3dev.ev3.Leds`

The EV3 LEDs.

EV3 platform

Led groups:

LEFT

RIGHT

Colors:

RED

GREEN

AMBER

ORANGE

YELLOW

BrickPI platform

Led groups:

LED1

LED2

Colors:

BLUE

static `all_off()`

Turn all leds off

static `set(group, **kwargs)`

Set attributes for each led in group.

Example:

```
Leds.set(LEFT, brightness_pct=0.5, trigger='timer')
```

static `set_color(group, color, pct=1)`

Sets brightness of leds in the given group to the values specified in color tuple. When percentage is specified, brightness of each led is reduced proportionally.

Example:

```
Leds.set_color(LEFT, AMBER)
```

Power Supply

class `ev3dev.core.PowerSupply` (*address=None*, *name_pattern='**, *name_exact=False*, ***kwargs*)

A generic interface to read data from the system's power_supply class. Uses the built-in legoev3-battery if none is specified.

max_voltage

measured_amps

The measured current that the battery is supplying (in amps)

measured_current

The measured current that the battery is supplying (in microamps)

measured_voltage

The measured voltage that the battery is supplying (in microvolts)

measured_volts

The measured voltage that the battery is supplying (in volts)

min_voltage

technology

type

Sound

class `ev3dev.core.Sound`

Sound-related functions. The class has only static methods and is not intended for instantiation. It can beep, play wav files, or convert text to speech.

Note that all methods of the class spawn system processes and return `subprocess.Popen` objects. The methods are asynchronous (they return immediately after child process was spawned, without waiting for its completion), but you can call `wait()` on the returned result.

Examples:

```
# Play 'bark.wav', return immediately:
Sound.play('bark.wav')

# Introduce yourself, wait for completion:
Sound.speak('Hello, I am Robot').wait()

# Play a small song
Sound.play_song((
    ('D4', 'e3'),
    ('D4', 'e3'),
    ('D4', 'e3'),
    ('G4', 'h'),
    ('D5', 'h')
))
```

static `beep` (*args=""*)

Call beep command with the provided arguments (if any). See [beep man page](#) and google [linux beep music](#) for inspiration.

static `get_volume` (*channel=None*)

Gets the current sound volume by parsing the output of `amixer get <channel>`. If the channel is

not specified, it tries to determine the default one by running `amixer scontrols`. If that fails as well, it uses the `Playback` channel, as that is the only channel on the EV3.

static play (*wav_file*)

Play wav file.

classmethod play_song (*song, tempo=120, delay=50*)

Plays a song provided as a list of tuples containing the note name and its value using music conventional notation instead of numerical values for frequency and duration.

It supports symbolic notes (e.g. A4, D#3, Gb5) and durations (e.g. q, h).

For an exhaustive list of accepted note symbols and values, have a look at the `_NOTE_FREQUENCIES` and `_NOTE_VALUES` private dictionaries in the source code.

The value can be suffixed by modifiers:

- a *divider* introduced by a / to obtain triplets for instance (e.g. q/3 for a triplet of eight note)
- a *multiplier* introduced by * (e.g. *1.5 is a dotted note).

Shortcuts exist for common modifiers:

- 3 produces a triplet member note. For instance e3 gives a triplet of eight notes, i.e. 3 eight notes in the duration of a single quarter. You must ensure that 3 triplets notes are defined in sequence to match the count, otherwise the result will not be the expected one.
- . produces a dotted note, i.e. which duration is one and a half the base one. Double dots are not currently supported.

Example:

```
>>> # A long time ago in a galaxy far,
>>> # far away...
>>> Sound.play_song((
>>>     ('D4', 'e3'),          # intro anacrouse
>>>     ('D4', 'e3'),
>>>     ('D4', 'e3'),
>>>     ('G4', 'h'),          # meas 1
>>>     ('D5', 'h'),
>>>     ('C5', 'e3'),          # meas 2
>>>     ('B4', 'e3'),
>>>     ('A4', 'e3'),
>>>     ('G5', 'h'),
>>>     ('D5', 'q'),
>>>     ('C5', 'e3'),          # meas 3
>>>     ('B4', 'e3'),
>>>     ('A4', 'e3'),
>>>     ('G5', 'h'),
>>>     ('D5', 'q'),
>>>     ('C5', 'e3'),          # meas 4
>>>     ('B4', 'e3'),
>>>     ('C5', 'e3'),
>>>     ('A4', 'h.'),
>>> ))
```

Important: Only 4/4 signature songs are supported with respect to note durations.

Args: song (iterable[tuple(str, str)]): the song tempo (int): the song tempo, given in quarters per minute
delay (int): delay in ms between notes

Returns: subprocess.Popen: the spawn subprocess

static set_volume (*pct, channel=None*)

Sets the sound volume to the given percentage [0-100] by calling `amixer -q set <channel> <pct>%`. If the channel is not specified, it tries to determine the default one by running `amixer scontrols`. If that fails as well, it uses the Playback channel, as that is the only channel on the EV3.

static speak (*text, espeak_opts='-a 200 -s 130'*)

Speak the given text aloud.

static tone (**args*)

tone(tone_sequence)

Play tone sequence. The `tone_sequence` parameter is a list of tuples, where each tuple contains up to three numbers. The first number is frequency in Hz, the second is duration in milliseconds, and the third is delay in milliseconds between this and the next tone in the sequence.

Here is a cheerful example:

```
Sound.tone([
    (392, 350, 100), (392, 350, 100), (392, 350, 100), (311.1, 250, 100),
    (466.2, 25, 100), (392, 350, 100), (311.1, 250, 100), (466.2, 25, 100),
    (392, 700, 100), (587.32, 350, 100), (587.32, 350, 100),
    (587.32, 350, 100), (622.26, 250, 100), (466.2, 25, 100),
    (369.99, 350, 100), (311.1, 250, 100), (466.2, 25, 100), (392, 700, 100),
    (784, 350, 100), (392, 250, 100), (392, 25, 100), (784, 350, 100),
    (739.98, 250, 100), (698.46, 25, 100), (659.26, 25, 100),
    (622.26, 25, 100), (659.26, 50, 400), (415.3, 25, 200), (554.36, 350, ↵
↵100),
    (523.25, 250, 100), (493.88, 25, 100), (466.16, 25, 100), (440, 25, 100),
    (466.16, 50, 400), (311.13, 25, 200), (369.99, 350, 100),
    (311.13, 250, 100), (392, 25, 100), (466.16, 350, 100), (392, 250, 100),
    (466.16, 25, 100), (587.32, 700, 100), (784, 350, 100), (392, 250, 100),
    (392, 25, 100), (784, 350, 100), (739.98, 250, 100), (698.46, 25, 100),
    (659.26, 25, 100), (622.26, 25, 100), (659.26, 50, 400), (415.3, 25, 200),
    (554.36, 350, 100), (523.25, 250, 100), (493.88, 25, 100),
    (466.16, 25, 100), (440, 25, 100), (466.16, 50, 400), (311.13, 25, 200),
    (392, 350, 100), (311.13, 250, 100), (466.16, 25, 100),
    (392.00, 300, 150), (311.13, 250, 100), (466.16, 25, 100), (392, 700)
]).wait()
```

tone(frequency, duration)

Play single tone of given frequency (Hz) and duration (milliseconds).

Screen

class ev3dev.core.Screen

Bases: ev3dev.core.FbMem

A convenience wrapper for the FbMem class. Provides drawing functions from the python imaging library (PIL).

clear()

Clears the screen

draw

Returns a handle to `PIL.ImageDraw.Draw` class associated with the screen.

Example:

```
screen.draw.rectangle((10,10,60,20), fill='black')
```

image

Returns a handle to `PIL.Image` class that is backing the screen. This can be accessed for blitting images to the screen.

Example:

```
screen.image.paste(picture, (0, 0))
```

shape

Dimensions of the screen.

update()

Applies pending changes to the screen. Nothing will be drawn on the screen until this function is called.

xres

Horizontal screen resolution

yres

Vertical screen resolution

Bitmap fonts

The `Screen` class allows to write text on the LCD using python imaging library (PIL) interface (see description of the `text()` method [here](#)). The `ev3dev.fonts` module contains bitmap fonts in PIL format that should look good on a tiny EV3 screen:

```
import ev3dev.fonts as fonts
screen.draw.text((10,10), 'Hello World!', font=fonts.load('luBS14'))
```

`ev3dev.fonts.available()`

Returns list of available font names.

`ev3dev.fonts.load(name)`

Loads the font specified by name and returns it as an instance of `PIL.ImageFont` class.

The following image lists all available fonts. The grid lines correspond to EV3 screen size:

charB08	charB10	charB12	charB14	charB18	charB24	charB108	charB110
charB112	charB114	charB118	charB124	charI08	charI10	charI12	charI14
charI18	charI24	charR08	charR10	charR12	charR14	charR18	charR24
courB08	courB10	courB12	courB14	courB18	courB24	courB008	courB010
courB012	courB014	courB018	courB024	courO08	courO10	courO12	courO14
courO18	courO24	courR08	courR10	courR12	courR14	courR18	courR24
helvB08	helvB10	helvB12	helvB14	helvB18	helvB24	helvB008	helvB010
helvB012	helvB014	helvB018	helvB024	helvO08	helvO10	helvO12	helvO14
helvO18	helvO24	helvR08	helvR10	helvR12	helvR14	helvR18	helvR24
luBS08	luBS10	luBS12	luBS14	luBS18	luBS19	luBS24	luBS08
luBS10	luBS12	luBS14	luBS18	luBS19	luBS24	luRS08	luRS10
luRS12	luRS14	luRS18	luRS19	luRS24	luRS08	luRS10	luRS12
luRS14	luRS18	luRS19	luRS24	lubB08	lubB10	lubB12	lubB14
lubB18	lubB19	lubB24	lubB08	lubB110	lubB112	lubB114	lubB118
lubB119	lubB124	lubB08	lubB110	lubB112	lubB114	lubB118	lubB119
lubB124	lubB08	lubB110	lubB112	lubB114	lubB118	lubB119	lubB124
lutBS08	lutBS10	lutBS12	lutBS14	lutBS18	lutBS19	lutBS24	lutBS08
lutRS10	lutRS12	lutRS14	lutRS18	lutRS19	lutRS24	ncenB08	ncenB10
ncenB12	ncenB14	ncenB18	ncenB24	ncenB108	ncenB110	ncenB112	ncenB114
ncenB118	ncenB124	ncenB08	ncenB110	ncenB112	ncenB114	ncenB118	ncenB124
ncenR08	ncenR10	ncenR12	ncenR14	ncenR18	ncenR24	σμβ08	σμβ10
σμβ12	σμβ14	σμβ18	σμβ24	term14	termB14	term14	termB14
timB08	timB10	timB12	timB14	timB18	timB24	timB08	timB110
timB112	timB114	timB118	timB124	timO08	timO10	timO12	timO14
timO18	timO24	timR08	timR10	timR12	timR14	timR18	timR24

Lego Port

class `ev3dev.core.LegoPort` (*address=None, name_pattern='*', name_exact=False, **kwargs*)

The *lego-port* class provides an interface for working with input and output ports that are compatible with LEGO MINDSTORMS RCX/NXT/EV3, LEGO WeDo and LEGO Power Functions sensors and motors. Supported devices include the LEGO MINDSTORMS EV3 Intelligent Brick, the LEGO WeDo USB hub and various sensor multiplexers from 3rd party manufacturers.

Some types of ports may have multiple modes of operation. For example, the input ports on the EV3 brick can communicate with sensors using UART, I2C or analog voltage signals - but not all at the same time. Therefore there are multiple modes available to connect to the different types of sensors.

In most cases, ports are able to automatically detect what type of sensor or motor is connected. In some cases though, this must be manually specified using the *mode* and *set_device* attributes. The *mode* attribute affects how the port communicates with the connected device. For example the input ports on the EV3 brick can communicate using UART, I2C or analog voltages, but not all at the same time, so the mode must be set to the one that is appropriate for the connected sensor. The *set_device* attribute is used to specify the exact type of sensor that is connected. Note: the mode must be correctly set before setting the sensor type.

Ports can be found at `/sys/class/lego-port/port<N>` where `<N>` is incremented each time a new port is registered. Note: The number is not related to the actual port at all - use the *address* attribute to find a specific port.

address

Returns the name of the port. See individual driver documentation for the name that will be returned.

driver_name

Returns the name of the driver that loaded this device. You can find the complete list of drivers in the [list of port drivers].

mode

Reading returns the currently selected mode. Writing sets the mode. Generally speaking when the mode changes any sensor or motor devices associated with the port will be removed new ones loaded, however this this will depend on the individual driver implementing this class.

modes

Returns a list of the available modes of the port.

set_device

For modes that support it, writing the name of a driver will cause a new device to be registered for that driver and attached to this port. For example, since NXT/Analog sensors cannot be auto-detected, you must use this attribute to load the correct driver. Returns `-EOPNOTSUPP` if setting a device is not supported.

status

In most cases, reading status will return the same value as *mode*. In cases where there is an *auto* mode additional values may be returned, such as *no-device* or *error*. See individual port driver documentation for the full list of possible values.

7.2 Working with ev3dev remotely using RPyC

RPyC (pronounced as are-pie-see), or Remote Python Call, is a transparent python library for symmetrical remote procedure calls, clustering and distributed-computing. RPyC makes use of object-proxying, a technique that employs python's dynamic nature, to overcome the physical boundaries between processes and computers, so that remote objects can be manipulated as if they were local. Here are simple steps you need to follow in order to install and use RPyC with ev3dev:

1. Install RPyC both on the EV3 and on your desktop PC. For the EV3, enter the following command at the command prompt (after you [connect with SSH](#)):

```
sudo easy_install3 rpyc
```

On the desktop PC, it really depends on your operating system. In case it is some flavor of linux, you should be able to do

```
sudo pip3 install rpyc
```

In case it is Windows, there is a win32 installer on the project's [sourceforge page](#). Also, have a look at the [Download and Install](#) page on their site.

2. Create file `rpyc_server.sh` with the following contents on the EV3:

```
#!/bin/bash
python3 `which rpyc_classic.py`
```

and make the file executable:

```
chmod +x rpyc_server.sh
```

Launch the created file either from SSH session (with `./rpyc_server.sh` command), or from brickman. It should output something like

```
INFO:SLAVE/18812:server started on [0.0.0.0]:18812
```

and keep running.

3. Now you are ready to connect to the RPyC server from your desktop PC. The following python script should make a large motor connected to output port A spin for a second.

```
import rpyc
conn = rpyc.classic.connect('ev3dev') # host name or IP address of the EV3
ev3 = conn.modules['ev3dev.ev3']     # import ev3dev.ev3 remotely
m = ev3.LargeMotor('outA')
m.run_timed(time_sp=1000, speed_sp=600)
```

You can run scripts like this from any interactive python environment, like ipython shell/notebook, spyder, pycharm, etc.

Some *advantages* of using RPyC with ev3dev are:

- It uses much less resources than running ipython notebook on EV3; RPyC server is lightweight, and only requires an IP connection to the EV3 once set up (no ssh required).
- The scripts you are working with are actually stored and edited on your desktop PC, with your favorite editor/IDE.
- Some robots may need much more computational power than what EV3 can give you. A notable example is the Rubics cube solver: there is an algorithm that provides almost optimal solution (in terms of number of cube rotations), but it takes more RAM than is available on EV3. With RPYC, you could run the heavy-duty computations on your desktop.

The most obvious *disadvantage* is latency introduced by network connection. This may be a show stopper for robots where reaction speed is essential.

7.3 Frequently-Asked Questions

7.3.1 My script works when launched as `python3 script.py` but exits immediately or throws an error when launched from Brickman or as `./script.py`

This may occur if your file includes Windows-style line endings, which are often inserted by editors on Windows. To resolve this issue, open an SSH session and run the following command, replacing `<file>` with the name of the Python file you're using:

```
sed -i 's/\r//g' <file>
```

This will fix it for the copy of the file on the brick, but if you plan to edit it again from Windows you should configure your editor to use Unix-style endings. For PyCharm, you can find a guide on doing this [here](#). Most other editors have similar options; there may be an option for it in the status bar at the bottom of the window or in the menu bar at the top.

CHAPTER 8

Indices and tables

- `genindex`
- `modindex`
- `search`

A

address (ev3dev.core.DcMotor attribute), 22
 address (ev3dev.core.LegoPort attribute), 39
 address (ev3dev.core.Motor attribute), 17
 address (ev3dev.core.Sensor attribute), 24
 address (ev3dev.core.ServoMotor attribute), 23
 all_off() (ev3dev.ev3.Leds static method), 33
 ambient_light_intensity (ev3dev.core.ColorSensor attribute), 26
 ambient_light_intensity (ev3dev.core.LightSensor attribute), 29
 angle (ev3dev.core.GyroSensor attribute), 28
 any() (ev3dev.core.RemoteControl method), 30
 any() (ev3dev.ev3.Button method), 31
 available() (in module ev3dev.fonts), 37

B

backspace (ev3dev.ev3.Button attribute), 31
 beacon (ev3dev.core.RemoteControl attribute), 30
 BeaconSeeker (class in ev3dev.core), 30
 beep() (ev3dev.core.Sound static method), 34
 bin_data() (ev3dev.core.Sensor method), 24
 bin_data_format (ev3dev.core.Sensor attribute), 24
 blue (ev3dev.core.ColorSensor attribute), 26
 blue_down (ev3dev.core.RemoteControl attribute), 30
 blue_up (ev3dev.core.RemoteControl attribute), 30
 brightness (ev3dev.core.Led attribute), 32
 brightness_pct (ev3dev.core.Led attribute), 32
 Button (class in ev3dev.ev3), 31
 Button.on_backspace (in module ev3dev.ev3), 31
 Button.on_down (in module ev3dev.ev3), 31
 Button.on_enter (in module ev3dev.ev3), 31
 Button.on_left (in module ev3dev.ev3), 31
 Button.on_right (in module ev3dev.ev3), 31
 Button.on_up (in module ev3dev.ev3), 31
 buttons_pressed (ev3dev.core.RemoteControl attribute), 30
 buttons_pressed (ev3dev.ev3.Button attribute), 31

C

check_buttons() (ev3dev.core.RemoteControl method), 30
 check_buttons() (ev3dev.ev3.Button method), 31
 clear() (ev3dev.core.Screen method), 36
 color (ev3dev.core.ColorSensor attribute), 26
 COLOR_BLACK (ev3dev.core.ColorSensor attribute), 26
 COLOR_BLUE (ev3dev.core.ColorSensor attribute), 26
 COLOR_BROWN (ev3dev.core.ColorSensor attribute), 26
 COLOR_GREEN (ev3dev.core.ColorSensor attribute), 26
 COLOR_NOCOLOR (ev3dev.core.ColorSensor attribute), 26
 COLOR_RED (ev3dev.core.ColorSensor attribute), 26
 COLOR_WHITE (ev3dev.core.ColorSensor attribute), 26
 COLOR_YELLOW (ev3dev.core.ColorSensor attribute), 26
 ColorSensor (class in ev3dev.core), 26
 command (ev3dev.core.DcMotor attribute), 22
 command (ev3dev.core.Motor attribute), 17
 command (ev3dev.core.Sensor attribute), 25
 command (ev3dev.core.ServoMotor attribute), 23
 COMMAND_FLOAT (ev3dev.core.ServoMotor attribute), 23
 COMMAND_RESET (ev3dev.core.Motor attribute), 16
 COMMAND_RUN (ev3dev.core.ServoMotor attribute), 23
 COMMAND_RUN_DIRECT (ev3dev.core.DcMotor attribute), 21
 COMMAND_RUN_DIRECT (ev3dev.core.Motor attribute), 16
 COMMAND_RUN_FOREVER (ev3dev.core.DcMotor attribute), 21
 COMMAND_RUN_FOREVER (ev3dev.core.Motor attribute), 16
 COMMAND_RUN_TIMED (ev3dev.core.DcMotor attribute), 21

COMMAND_RUN_TIMED (ev3dev.core.Motor attribute), 16
COMMAND_RUN_TO_ABS_POS (ev3dev.core.Motor attribute), 17
COMMAND_RUN_TO_REL_POS (ev3dev.core.Motor attribute), 17
COMMAND_STOP (ev3dev.core.DcMotor attribute), 21
COMMAND_STOP (ev3dev.core.Motor attribute), 17
commands (ev3dev.core.DcMotor attribute), 22
commands (ev3dev.core.Motor attribute), 17
commands (ev3dev.core.Sensor attribute), 25
count_per_m (ev3dev.core.Motor attribute), 18
count_per_rot (ev3dev.core.Motor attribute), 18

D

DcMotor (class in ev3dev.core), 21
decimals (ev3dev.core.Sensor attribute), 25
delay_off (ev3dev.core.Led attribute), 32
delay_on (ev3dev.core.Led attribute), 32
Device (class in ev3dev.core), 15
distance (ev3dev.core.BeaconSeeker attribute), 30
distance_centimeters (ev3dev.core.UltrasonicSensor attribute), 27
distance_inches (ev3dev.core.UltrasonicSensor attribute), 27
down (ev3dev.ev3.Button attribute), 31
draw (ev3dev.core.Screen attribute), 37
driver_name (ev3dev.core.DcMotor attribute), 22
driver_name (ev3dev.core.LegoPort attribute), 39
driver_name (ev3dev.core.Motor attribute), 18
driver_name (ev3dev.core.Sensor attribute), 25
driver_name (ev3dev.core.ServoMotor attribute), 23
duty_cycle (ev3dev.core.DcMotor attribute), 22
duty_cycle (ev3dev.core.Motor attribute), 18
duty_cycle_sp (ev3dev.core.DcMotor attribute), 22
duty_cycle_sp (ev3dev.core.Motor attribute), 18

E

ENCODER_POLARITY_INVERSED (ev3dev.core.Motor attribute), 17
ENCODER_POLARITY_NORMAL (ev3dev.core.Motor attribute), 17
enter (ev3dev.ev3.Button attribute), 31

F

float() (ev3dev.core.ServoMotor method), 23
full_travel_count (ev3dev.core.Motor attribute), 18

G

get_volume() (ev3dev.core.Sound static method), 34
green (ev3dev.core.ColorSensor attribute), 27
GyroSensor (class in ev3dev.core), 27

H

heading (ev3dev.core.BeaconSeeker attribute), 30
heading_and_distance (ev3dev.core.BeaconSeeker attribute), 30

I

image (ev3dev.core.Screen attribute), 37
InfraredSensor (class in ev3dev.core), 28
is_holding (ev3dev.core.Motor attribute), 18
is_overloaded (ev3dev.core.Motor attribute), 18
is_pressed (ev3dev.core.TouchSensor attribute), 25
is_ramping (ev3dev.core.Motor attribute), 18
is_running (ev3dev.core.Motor attribute), 18
is_stalled (ev3dev.core.Motor attribute), 18

L

LargeMotor (class in ev3dev.core), 21
Led (class in ev3dev.core), 32
Leds (class in ev3dev.ev3), 33
Leds.AMBER (in module ev3dev.ev3), 33
Leds.BLUE (in module ev3dev.ev3), 33
Leds.GREEN (in module ev3dev.ev3), 33
Leds.LED1 (in module ev3dev.ev3), 33
Leds.LED2 (in module ev3dev.ev3), 33
Leds.LEFT (in module ev3dev.ev3), 33
Leds.ORANGE (in module ev3dev.ev3), 33
Leds.RED (in module ev3dev.ev3), 33
Leds.RIGHT (in module ev3dev.ev3), 33
Leds.YELLOW (in module ev3dev.ev3), 33
left (ev3dev.ev3.Button attribute), 31
LegoPort (class in ev3dev.core), 39
LightSensor (class in ev3dev.core), 29
list_device_names() (in module ev3dev.core), 15
list_devices() (in module ev3dev.core), 15
list_motors() (in module ev3dev.core), 16
list_sensors() (in module ev3dev.core), 16
load() (in module ev3dev.fonts), 37

M

max_brightness (ev3dev.core.Led attribute), 32
max_pulse_sp (ev3dev.core.ServoMotor attribute), 23
max_speed (ev3dev.core.Motor attribute), 18
max_voltage (ev3dev.core.PowerSupply attribute), 34
measured_amps (ev3dev.core.PowerSupply attribute), 34
measured_current (ev3dev.core.PowerSupply attribute), 34
measured_voltage (ev3dev.core.PowerSupply attribute), 34
measured_volts (ev3dev.core.PowerSupply attribute), 34
MediumMotor (class in ev3dev.core), 21
mid_pulse_sp (ev3dev.core.ServoMotor attribute), 23
min_pulse_sp (ev3dev.core.ServoMotor attribute), 23
min_voltage (ev3dev.core.PowerSupply attribute), 34

[mode \(ev3dev.core.LegoPort attribute\), 39](#)
[mode \(ev3dev.core.Sensor attribute\), 25](#)
[MODE_AMBIENT \(ev3dev.core.LightSensor attribute\), 29](#)
[MODE_COL_AMBIENT \(ev3dev.core.ColorSensor attribute\), 26](#)
[MODE_COL_COLOR \(ev3dev.core.ColorSensor attribute\), 26](#)
[MODE_COL_REFLECT \(ev3dev.core.ColorSensor attribute\), 26](#)
[MODE_DB \(ev3dev.core.SoundSensor attribute\), 28](#)
[MODE_DBA \(ev3dev.core.SoundSensor attribute\), 29](#)
[MODE_GYRO_ANG \(ev3dev.core.GyroSensor attribute\), 28](#)
[MODE_GYRO_CAL \(ev3dev.core.GyroSensor attribute\), 28](#)
[MODE_GYRO_FAS \(ev3dev.core.GyroSensor attribute\), 28](#)
[MODE_GYRO_G_A \(ev3dev.core.GyroSensor attribute\), 28](#)
[MODE_GYRO_RATE \(ev3dev.core.GyroSensor attribute\), 28](#)
[MODE_IR_CAL \(ev3dev.core.InfraredSensor attribute\), 28](#)
[MODE_IR_PROX \(ev3dev.core.InfraredSensor attribute\), 28](#)
[MODE_IR_REM_A \(ev3dev.core.InfraredSensor attribute\), 28](#)
[MODE_IR_REMOTE \(ev3dev.core.InfraredSensor attribute\), 28](#)
[MODE_IR_SEEK \(ev3dev.core.InfraredSensor attribute\), 28](#)
[MODE_REF_RAW \(ev3dev.core.ColorSensor attribute\), 26](#)
[MODE_REFLECT \(ev3dev.core.LightSensor attribute\), 29](#)
[MODE_RGB_RAW \(ev3dev.core.ColorSensor attribute\), 26](#)
[MODE_TOUCH \(ev3dev.core.TouchSensor attribute\), 25](#)
[MODE_US_DIST_CM \(ev3dev.core.UltrasonicSensor attribute\), 27](#)
[MODE_US_DIST_IN \(ev3dev.core.UltrasonicSensor attribute\), 27](#)
[MODE_US_LISTEN \(ev3dev.core.UltrasonicSensor attribute\), 27](#)
[MODE_US_SI_CM \(ev3dev.core.UltrasonicSensor attribute\), 27](#)
[MODE_US_SI_IN \(ev3dev.core.UltrasonicSensor attribute\), 27](#)
[modes \(ev3dev.core.LegoPort attribute\), 39](#)
[modes \(ev3dev.core.Sensor attribute\), 25](#)
[Motor \(class in ev3dev.core\), 16](#)

N

[num_values \(ev3dev.core.Sensor attribute\), 25](#)

O

[on_backspace\(\) \(ev3dev.ev3.Button static method\), 31](#)
[on_beacon \(ev3dev.core.RemoteControl attribute\), 30](#)
[on_blue_down \(ev3dev.core.RemoteControl attribute\), 30](#)
[on_blue_up \(ev3dev.core.RemoteControl attribute\), 30](#)
[on_change\(\) \(ev3dev.core.RemoteControl static method\), 30](#)
[on_change\(\) \(ev3dev.ev3.Button static method\), 31](#)
[on_down\(\) \(ev3dev.ev3.Button static method\), 31](#)
[on_enter\(\) \(ev3dev.ev3.Button static method\), 31](#)
[on_left\(\) \(ev3dev.ev3.Button static method\), 32](#)
[on_red_down \(ev3dev.core.RemoteControl attribute\), 30](#)
[on_red_up \(ev3dev.core.RemoteControl attribute\), 30](#)
[on_right\(\) \(ev3dev.ev3.Button static method\), 32](#)
[on_up\(\) \(ev3dev.ev3.Button static method\), 32](#)
[other_sensor_present \(ev3dev.core.UltrasonicSensor attribute\), 27](#)

P

[play\(\) \(ev3dev.core.Sound static method\), 35](#)
[play_song\(\) \(ev3dev.core.Sound class method\), 35](#)
[polarity \(ev3dev.core.DcMotor attribute\), 22](#)
[polarity \(ev3dev.core.Motor attribute\), 18](#)
[polarity \(ev3dev.core.ServoMotor attribute\), 23](#)
[POLARITY_INVERSED \(ev3dev.core.DcMotor attribute\), 21](#)
[POLARITY_INVERSED \(ev3dev.core.Motor attribute\), 17](#)
[POLARITY_INVERSED \(ev3dev.core.ServoMotor attribute\), 23](#)
[POLARITY_NORMAL \(ev3dev.core.DcMotor attribute\), 21](#)
[POLARITY_NORMAL \(ev3dev.core.Motor attribute\), 17](#)
[POLARITY_NORMAL \(ev3dev.core.ServoMotor attribute\), 23](#)
[position \(ev3dev.core.Motor attribute\), 19](#)
[position_d \(ev3dev.core.Motor attribute\), 19](#)
[position_i \(ev3dev.core.Motor attribute\), 19](#)
[position_p \(ev3dev.core.Motor attribute\), 19](#)
[position_sp \(ev3dev.core.Motor attribute\), 19](#)
[position_sp \(ev3dev.core.ServoMotor attribute\), 23](#)
[PowerSupply \(class in ev3dev.core\), 34](#)
[process\(\) \(ev3dev.core.RemoteControl method\), 30](#)
[process\(\) \(ev3dev.ev3.Button method\), 32](#)
[proximity \(ev3dev.core.InfraredSensor attribute\), 28](#)

R

[ramp_down_sp \(ev3dev.core.DcMotor attribute\), 22](#)
[ramp_down_sp \(ev3dev.core.Motor attribute\), 19](#)

ramp_up_sp (ev3dev.core.DcMotor attribute), 22
ramp_up_sp (ev3dev.core.Motor attribute), 19
rate (ev3dev.core.GyroSensor attribute), 28
rate_and_angle (ev3dev.core.GyroSensor attribute), 28
rate_sp (ev3dev.core.ServoMotor attribute), 24
raw (ev3dev.core.ColorSensor attribute), 27
red (ev3dev.core.ColorSensor attribute), 27
red_down (ev3dev.core.RemoteControl attribute), 30
red_up (ev3dev.core.RemoteControl attribute), 30
reflected_light_intensity (ev3dev.core.ColorSensor attribute), 27
reflected_light_intensity (ev3dev.core.LightSensor attribute), 29
RemoteControl (class in ev3dev.core), 29
RemoteControl.on_beacon (in module ev3dev.core), 29
RemoteControl.on_blue_down (in module ev3dev.core), 29
RemoteControl.on_blue_up (in module ev3dev.core), 29
RemoteControl.on_red_down (in module ev3dev.core), 29
RemoteControl.on_red_up (in module ev3dev.core), 29
reset() (ev3dev.core.Motor method), 19
right (ev3dev.ev3.Button attribute), 32
run() (ev3dev.core.ServoMotor method), 24
run_direct() (ev3dev.core.DcMotor method), 22
run_direct() (ev3dev.core.Motor method), 19
run_forever() (ev3dev.core.DcMotor method), 22
run_forever() (ev3dev.core.Motor method), 19
run_timed() (ev3dev.core.DcMotor method), 22
run_timed() (ev3dev.core.Motor method), 19
run_to_abs_pos() (ev3dev.core.Motor method), 19
run_to_rel_pos() (ev3dev.core.Motor method), 19

S

Screen (class in ev3dev.core), 36
Sensor (class in ev3dev.core), 24
ServoMotor (class in ev3dev.core), 23
set() (ev3dev.ev3.Leds static method), 33
set_color() (ev3dev.ev3.Leds static method), 33
set_device (ev3dev.core.LegoPort attribute), 39
set_volume() (ev3dev.core.Sound static method), 36
shape (ev3dev.core.Screen attribute), 37
Sound (class in ev3dev.core), 34
sound_pressure (ev3dev.core.SoundSensor attribute), 29
sound_pressure_low (ev3dev.core.SoundSensor attribute), 29
SoundSensor (class in ev3dev.core), 28
speak() (ev3dev.core.Sound static method), 36
speed (ev3dev.core.Motor attribute), 19
speed_d (ev3dev.core.Motor attribute), 19
speed_i (ev3dev.core.Motor attribute), 20
speed_p (ev3dev.core.Motor attribute), 20
speed_sp (ev3dev.core.Motor attribute), 20
state (ev3dev.core.DcMotor attribute), 22

state (ev3dev.core.Motor attribute), 20
state (ev3dev.core.ServoMotor attribute), 24
STATE_HOLDING (ev3dev.core.Motor attribute), 17
STATE_OVERLOADED (ev3dev.core.Motor attribute), 17
STATE_RAMPING (ev3dev.core.Motor attribute), 17
STATE_RUNNING (ev3dev.core.Motor attribute), 17
STATE_STALLED (ev3dev.core.Motor attribute), 17
status (ev3dev.core.LegoPort attribute), 39
stop() (ev3dev.core.DcMotor method), 22
stop() (ev3dev.core.Motor method), 20
stop_action (ev3dev.core.DcMotor attribute), 22
stop_action (ev3dev.core.Motor attribute), 20
STOP_ACTION_BRAKE (ev3dev.core.DcMotor attribute), 21
STOP_ACTION_BRAKE (ev3dev.core.Motor attribute), 17
STOP_ACTION_COAST (ev3dev.core.DcMotor attribute), 22
STOP_ACTION_COAST (ev3dev.core.Motor attribute), 17
STOP_ACTION_HOLD (ev3dev.core.Motor attribute), 17
stop_actions (ev3dev.core.DcMotor attribute), 22
stop_actions (ev3dev.core.Motor attribute), 20

T

technology (ev3dev.core.PowerSupply attribute), 34
time_sp (ev3dev.core.DcMotor attribute), 23
time_sp (ev3dev.core.Motor attribute), 20
tone() (ev3dev.core.Sound static method), 36
TouchSensor (class in ev3dev.core), 25
trigger (ev3dev.core.Led attribute), 32
triggers (ev3dev.core.Led attribute), 32
type (ev3dev.core.PowerSupply attribute), 34

U

UltrasonicSensor (class in ev3dev.core), 27
units (ev3dev.core.Sensor attribute), 25
up (ev3dev.ev3.Button attribute), 32
update() (ev3dev.core.Screen method), 37

V

value() (ev3dev.core.Sensor method), 25

W

wait() (ev3dev.core.Motor method), 20
wait_until() (ev3dev.core.Motor method), 20
wait_until_not_moving() (ev3dev.core.Motor method), 20
wait_while() (ev3dev.core.Motor method), 21

X

xres (ev3dev.core.Screen attribute), 37

Y

yres (ev3dev.core.Screen attribute), [37](#)