python-ev3dev Documentation

Release 0.6.0.post64

Ralph Hempel et al

May 27, 2016

Contents

1 Example Code 3

2 User Resources 5

3 Developer Resources 7

4 Python2.x and Python3.x Compatibility 9
- 4.1 API reference . . . 9
  - 4.1.1 Motor classes . . . 10
  - 4.1.2 Sensor classes . . . 15
  - 4.1.3 Other classes . . . 19

5 Indices and tables 25

A Python library implementing unified interface for ev3dev devices.

CHAPTER 1

Example Code

To run these minimal examples, run the Python interpreter from the terminal like this:

```
robot@ev3dev:~$ python
Python 2.7.9 (default, Mar  1 2015, 13:52:09)
[GCC 4.9.2] on linux2
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.

Load the ev3dev-lang bindings:

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

Now let's try our first program. 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()])
```

Now plug a motor into the `A` port and paste this code into the terminal. This little program will run the motor at 75% power for 3 seconds.

```
m = ev3.LargeMotor('outA')
m.run_timed(time_sp=3000, duty_cycle_sp=75)
```

If you want to make your robot speak, then paste this code into the terminal:

```
ev3.Sound.speak('Welcome to the EV3DEV project!').wait()
```

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

CHAPTER 2

User Resources

Getting Started with ev3dev If you got here as the result of looking for "how to program LEGO MINDSTORMS EV3 using Python" then you might not be aware that this is part of a much larger project called ev3dev. Make sure you read the Getting Started page to become familiar with ev3dev first!

Connecting the EV3 to the Internet You can connect to an EV3 running ev3dev using USB, Wifi or Bluetooth. The USB connection is a good starting point, and the ev3dev site has detailed instructions for USB connections for Linux, Windows, and Mac computers.

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 3

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.

CHAPTER 4

Python2.x and Python3.x Compatibility

The ev3dev distribution comes with both python2 and python3 installed and this library is compatible with both versions.

Note that currently, the source is only installed in the default Python 2.x location - this will be addressed in the next package we release.

For Python 2.x programs, you import the binding like this:

```
from ev3dev.auto import *
```

For Python 3.x the easiest way to work around the problem is to get your EV3 connected to the Internet and then:

1. Update the package lists
2. Install the `python3-pil` package
3. Use `easy-install` install `python-ev3dev`

```
sudo apt-get update
sudo apt-get install python3-pil
sudo python3 -m easy_install python-ev3dev
```

You will be asked for the `robot` user’s password to get `sudo` access to the system - the default password is `maker`.

Please be patient - a typical `apt-get update` will take about 10 minutes - there’s a LOT going on under the hood to sort out package dependencies.

And now you can use ev3dev-lang-python under Python 3.x.

```
from ev3dev.auto import *
```

Contents

4.1 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

Contents:

4.1.1 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.

`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 command specified in *stop_command*.
- *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 command specified by *stop_command*.
- *run-timed* will run the motor for the amount of time specified in *time_sp* and then stop the motor using the command specified by *stop_command*.
- *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 command specified by *stop_command*.
- *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* atribute, you can use this value to calculate the maximum travel distance of the motor. (linear motors only)

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 command

specified by *stop_command*.

run_to_abs_pos(***kwargs*)
Run to an absolute position specified by *position_sp* and then stop using the command specified in *stop_command*.

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 command specified by *stop_command*.

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* and *stalled*.

stop(***kwargs*)
Stop any of the run commands before they are complete using the command specified by *stop_command*.

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.

Large EV3 Motor

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

EV3 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.

`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 command specified by *stop_command*.

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 command specified by *stop_command*.

stop_command
Sets the stop command that will be used when the motor stops. Read *stop_commands* to get the list of valid values.

stop_commands
Gets a list of stop commands. 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.

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 miniumum (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.

4.1.2 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

`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.

`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.

`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.

`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.

`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.

Infrared Sensor

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

Bases: `ev3dev.core.Sensor`

LEGO EV3 infrared sensor.

`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

`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

`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.

4.1.3 Other classes

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 brigthness 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

Button

class ev3dev.ev3.**Button**
EV3 Buttons

`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.

`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 currenly 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.

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()

static **`beep`**(*args=’‘*)

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

static **`play`**(*wav_file*)

Play wav file.

static **`speak`**(*text*)

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')
```

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

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

A

address (ev3dev.core.DcMotor attribute), 13
address (ev3dev.core.Motor attribute), 10
address (ev3dev.core.Sensor attribute), 15
address (ev3dev.core.ServoMotor attribute), 14
all_off() (ev3dev.ev3.Leds static method), 20
ambient_light_intensity() (ev3dev.core.ColorSensor method), 17
ambient_light_intensity() (ev3dev.core.LightSensor method), 18
angle() (ev3dev.core.GyroSensor method), 18
any() (ev3dev.ev3.Button method), 20

B

backspace (ev3dev.ev3.Button attribute), 21
beep() (ev3dev.core.Sound static method), 22
bin_data() (ev3dev.core.Sensor method), 15
bin_data_format (ev3dev.core.Sensor attribute), 15
blue() (ev3dev.core.ColorSensor method), 17
brightness (ev3dev.core.Led attribute), 19
brightness_pct (ev3dev.core.Led attribute), 19
Button (class in ev3dev.ev3), 20
buttons_pressed (ev3dev.ev3.Button attribute), 21

C

check_buttons() (ev3dev.ev3.Button method), 21
clear() (ev3dev.core.Screen method), 23
color() (ev3dev.core.ColorSensor method), 17
ColorSensor (class in ev3dev.core), 17
command (ev3dev.core.DcMotor attribute), 13
command (ev3dev.core.Motor attribute), 10
command (ev3dev.core.Sensor attribute), 16
command (ev3dev.core.ServoMotor attribute), 14
commands (ev3dev.core.DcMotor attribute), 13
commands (ev3dev.core.Motor attribute), 10
commands (ev3dev.core.Sensor attribute), 16
count_per_m (ev3dev.core.Motor attribute), 10
count_per_rot (ev3dev.core.Motor attribute), 10

D

DcMotor (class in ev3dev.core), 13
decimals (ev3dev.core.Sensor attribute), 16
delay_off (ev3dev.core.Led attribute), 19
delay_on (ev3dev.core.Led attribute), 19
Device (class in ev3dev.core), 9
distance_centimeters() (ev3dev.core.UltrasonicSensor method), 17
distance_inches() (ev3dev.core.UltrasonicSensor method), 17
down (ev3dev.ev3.Button attribute), 21
draw (ev3dev.core.Screen attribute), 23
driver_name (ev3dev.core.DcMotor attribute), 13
driver_name (ev3dev.core.Motor attribute), 10
driver_name (ev3dev.core.Sensor attribute), 16
driver_name (ev3dev.core.ServoMotor attribute), 14
duty_cycle (ev3dev.core.DcMotor attribute), 13
duty_cycle (ev3dev.core.Motor attribute), 10
duty_cycle_sp (ev3dev.core.DcMotor attribute), 13
duty_cycle_sp (ev3dev.core.Motor attribute), 11

E

enter (ev3dev.ev3.Button attribute), 21

F

float() (ev3dev.core.ServoMotor method), 14
full_travel_count (ev3dev.core.Motor attribute), 11

G

green() (ev3dev.core.ColorSensor method), 17
GyroSensor (class in ev3dev.core), 18

I

InfraredSensor (class in ev3dev.core), 18
is_pressed() (ev3dev.core.TouchSensor method), 16

L

LargeMotor (class in ev3dev.core), 12
Led (class in ev3dev.core), 19
Leds (class in ev3dev.ev3), 19

Leds.AMBER (in module ev3dev.core), 19
Leds.BLUE (in module ev3dev.core), 20
Leds.GREEN (in module ev3dev.core), 19
Leds.LED1 (in module ev3dev.core), 20
Leds.LED2 (in module ev3dev.core), 20
Leds.LEFT (in module ev3dev.core), 19
Leds.ORANGE (in module ev3dev.core), 19
Leds.RED (in module ev3dev.core), 19
Leds.RIGHT (in module ev3dev.core), 19
Leds.YELLOW (in module ev3dev.core), 19
left (ev3dev.ev3.Button attribute), 21
LightSensor (class in ev3dev.core), 18

M

max_brightness (ev3dev.core.Led attribute), 19
max_pulse_sp (ev3dev.core.ServoMotor attribute), 14
max_speed (ev3dev.core.Motor attribute), 11
max_voltage (ev3dev.core.PowerSupply attribute), 20
measured_amps (ev3dev.core.PowerSupply attribute), 20
measured_current (ev3dev.core.PowerSupply attribute), 20
measured_voltage (ev3dev.core.PowerSupply attribute), 20
measured_volts (ev3dev.core.PowerSupply attribute), 20
MediumMotor (class in ev3dev.core), 13
mid_pulse_sp (ev3dev.core.ServoMotor attribute), 14
min_pulse_sp (ev3dev.core.ServoMotor attribute), 14
min_voltage (ev3dev.core.PowerSupply attribute), 20
mode (ev3dev.core.Sensor attribute), 16
modes (ev3dev.core.Sensor attribute), 16
Motor (class in ev3dev.core), 10

N

num_values (ev3dev.core.Sensor attribute), 16

O

on_backspace() (ev3dev.ev3.Button static method), 21
on_change() (ev3dev.ev3.Button method), 21
on_down() (ev3dev.ev3.Button static method), 21
on_enter() (ev3dev.ev3.Button static method), 21
on_left() (ev3dev.ev3.Button static method), 21
on_right() (ev3dev.ev3.Button static method), 21
on_up() (ev3dev.ev3.Button static method), 21
other_sensor_present() (ev3dev.core.UltrasonicSensor method), 17

P

play() (ev3dev.core.Sound static method), 22
polarity (ev3dev.core.DcMotor attribute), 13
polarity (ev3dev.core.Motor attribute), 11
polarity (ev3dev.core.ServoMotor attribute), 14
position (ev3dev.core.Motor attribute), 11
position_d (ev3dev.core.Motor attribute), 11
position_i (ev3dev.core.Motor attribute), 11
position_p (ev3dev.core.Motor attribute), 11
position_sp (ev3dev.core.Motor attribute), 11
position_sp (ev3dev.core.ServoMotor attribute), 14
PowerSupply (class in ev3dev.core), 20
process() (ev3dev.ev3.Button method), 21
proximity() (ev3dev.core.InfraredSensor method), 18

R

ramp_down_sp (ev3dev.core.DcMotor attribute), 13
ramp_down_sp (ev3dev.core.Motor attribute), 11
ramp_up_sp (ev3dev.core.DcMotor attribute), 13
ramp_up_sp (ev3dev.core.Motor attribute), 11
rate() (ev3dev.core.GyroSensor method), 18
rate_sp (ev3dev.core.ServoMotor attribute), 15
red() (ev3dev.core.ColorSensor method), 17
reflected_light_intensity() (ev3dev.core.ColorSensor method), 17
reflected_light_intensity() (ev3dev.core.LightSensor method), 18
reset() (ev3dev.core.Motor method), 11
right (ev3dev.ev3.Button attribute), 21
run() (ev3dev.core.ServoMotor method), 15
run_direct() (ev3dev.core.DcMotor method), 13
run_direct() (ev3dev.core.Motor method), 11
run_forever() (ev3dev.core.DcMotor method), 13
run_forever() (ev3dev.core.Motor method), 11
run_timed() (ev3dev.core.DcMotor method), 13
run_timed() (ev3dev.core.Motor method), 11
run_to_abs_pos() (ev3dev.core.Motor method), 12
run_to_rel_pos() (ev3dev.core.Motor method), 12

S

Screen (class in ev3dev.core), 23
Sensor (class in ev3dev.core), 15
ServoMotor (class in ev3dev.core), 14
set() (ev3dev.ev3.Leds static method), 20
set_color() (ev3dev.ev3.Leds static method), 20
shape (ev3dev.core.Screen attribute), 23
Sound (class in ev3dev.core), 22
sound_pressure() (ev3dev.core.SoundSensor method), 18
sound_pressure_low() (ev3dev.core.SoundSensor method), 18
SoundSensor (class in ev3dev.core), 18
speak() (ev3dev.core.Sound static method), 22
speed (ev3dev.core.Motor attribute), 12
speed_d (ev3dev.core.Motor attribute), 12
speed_i (ev3dev.core.Motor attribute), 12
speed_p (ev3dev.core.Motor attribute), 12
speed_sp (ev3dev.core.Motor attribute), 12
state (ev3dev.core.DcMotor attribute), 13
state (ev3dev.core.Motor attribute), 12
state (ev3dev.core.ServoMotor attribute), 15
stop() (ev3dev.core.DcMotor method), 14

stop() (ev3dev.core.Motor method), 12
stop_action (ev3dev.core.Motor attribute), 12
stop_actions (ev3dev.core.Motor attribute), 12
stop_command (ev3dev.core.DcMotor attribute), 14
stop_commands (ev3dev.core.DcMotor attribute), 14

T

technology (ev3dev.core.PowerSupply attribute), 20
time_sp (ev3dev.core.DcMotor attribute), 14
time_sp (ev3dev.core.Motor attribute), 12
tone() (ev3dev.core.Sound static method), 22
TouchSensor (class in ev3dev.core), 16
trigger (ev3dev.core.Led attribute), 19
triggers (ev3dev.core.Led attribute), 19
type (ev3dev.core.PowerSupply attribute), 20

U

UltrasonicSensor (class in ev3dev.core), 17
units (ev3dev.core.Sensor attribute), 16
up (ev3dev.ev3.Button attribute), 21
update() (ev3dev.core.Screen method), 23

V

value() (ev3dev.core.Sensor method), 16

X

xres (ev3dev.core.Screen attribute), 23

Y

yres (ev3dev.core.Screen attribute), 23