
ev3dev-lang Documentation

Release 1.0.0

The ev3dev team

January 17, 2016

1	ev3dev Language Wrapper Specification	3
1.1	Classes	4
1.2	Special sensor classes	14
1.3	Constants / Enums	20
2	Autogen scripts for ev3dev language bindings	23
2.1	Usage	23
2.2	How it works	23
2.3	Implementation Notes	24
2.4	Contributing to the autogen script	24

This repository stores the utilities and metadata information for the ev3dev-lang family of libraries. These libraries are easy-to-use interfaces for the APIs that are available on **ev3dev**-based devices.

The complete documentation is hosted at <http://ev3dev-lang.readthedocs.org>.

We support multiple libraries for various programming languages using a centralized “specification” which we keep updated as kernel changes are made. We don’t have the actual library code here (see below) – instead we use this repo to facilitate the maintenance of our bindings.

We currently support libraries for the following languages:

- **C++**
- **Node.js**
- **Python**

Each binding is written based on our central spec, so each has a uniform interface which is kept close to the ev3dev API surface while still encouraging language-specific enhancements.

Contents

ev3dev Language Wrapper Specification

This is an unofficial specification that defines a unified interface for language wrappers to expose the `ev3dev` device APIs.

General Notes

Because this specification is meant to be implemented in multiple languages, the specific naming conventions of properties, methods and classes are not defined here. Depending on the language, names will be slightly different (ex. “touchSensor” or “TouchSensor” or “touch-sensor”) so that they fit the language’s naming conventions.

Some concepts that apply to multiple classes are described as “abstracts”. These abstract sections explain how the class should handle specific situations, and do not necessarily translate in to their own class in the wrapper.

Implementation Notes (important)

- **File access.** There should be one class that is used or inherited from in all other classes that need to access object properties via file I/O. This class should check paths for validity, do basic error checking, and generally implement as much of the core I/O functionality as possible.
- **Errors.** All file access and other error-prone calls should be wrapped with error handling. If an error thrown by an external call is fatal, the wrapper should throw an error for the caller that states the error and gives some insight in to what actually happened.
- **Naming conventions.** All names should follow the language’s naming conventions. Keep the names consistent, so that users can easily find what they want.
- **Attribute types.** `int` and `string` attributes are read-write files containing a single value that is representable either as an integer or as a single word. A `string array` attribute is a readonly file that contains space-separated list of words, where each word is a possible value of some other `string` attribute. And a `string selector` attribute is a read-write file that contains space-separated list of possible values, where the currently selected value is enclosed in square brackets. Another value may be selected by writing a single word to the file.

Contents

1.1 Classes

1.1.1 Device (abstract)

class **Device**

This is the base class that handles control tasks for a single port or index. The class must chose one device out of the available ports to control. Given an IO port (in the constructor), an implementation should:

- If the specified port is blank or unspecified/undefined/null, the available devices should be enumerated until a suitable device is found. Any device is suitable when it's type is known to be compatible with the controlling class, and it meets any other requirements specified by the caller.
- If the specified port name is not blank, the available devices should be enumerated until a device is found that is plugged in to the specified port. The supplied port name should be compared directly to the value from the file, so that advanced port strings will match, such as `in1:mux3`.

If an error occurs after the initial connection, an exception should be thrown by the binding informing the caller of what went wrong. Unless the error is fatal to the application, no other actions should be taken.

connected

If a valid device is found while enumerating the ports, the `connected` variable is set to `true` (by default, it should be false). If `connected` is false when an attempt is made to read from or write to a property file, an error should be thrown (except while in the constructor).

1.1.2 Motor

class **Motor**

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*.

ev3dev docs link: <http://www.ev3dev.org/docs/drivers/tacho-motor-class/>

System properties

Address

string, read

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

Command

string, write

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

Commands

string array, read

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_Rot

int, read

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. In the case of linear actuators, the units here will be counts per centimeter.

Driver_Name

string, read

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

Duty_Cycle

int, read

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

Duty_Cycle_SP

int, read/write

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. This value is only used when *speed_regulation* is off.

Encoder_Polarity

string, read/write

Sets the polarity of the rotary encoder. This is an advanced feature to all use of motors that send inversed encoder signals to the EV3. This should be set correctly by the driver of a device. It You only need to change this value if you are using a unsupported device. Valid values are *normal* and *inversed*.

Polarity

string, read/write

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

int, read/write

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_P

int, read/write

The proportional constant for the position PID.

Position_I

int, read/write

The integral constant for the position PID.

Position_D

int, read/write

The derivative constant for the position PID.

Position_SP

int, read/write

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.

Speed

int, read

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_SP

int, read/write

Writing sets the target speed in tacho counts per second used when *speed_regulation* is on. Reading returns the current value. Use the *count_per_rot* attribute to convert RPM or deg/sec to tacho counts per second.

Ramp_Up_SP

int, read/write

Writing sets the ramp up setpoint. Reading returns the current value. Units are in milliseconds. When set to a value > 0, the motor will ramp the power sent to the motor from 0 to 100% duty cycle over the span of this setpoint when starting the motor. If the maximum duty cycle is limited by *duty_cycle_sp* or speed regulation, the actual ramp time duration will be less than the setpoint.

Ramp_Down_SP

int, read/write

Writing sets the ramp down setpoint. Reading returns the current value. Units are in milliseconds. When set to a value > 0, the motor will ramp the power sent to the motor from 100% duty cycle down to 0 over the span of this setpoint when stopping the motor. If the starting duty cycle is less than 100%, the ramp time duration will be less than the full span of the setpoint.

Speed_Regulation_Enabled

string, read/write

Turns speed regulation on or off. If speed regulation is on, the motor controller will vary the power supplied to the motor to try to maintain the speed specified in *speed_sp*. If speed regulation is off, the controller will use the power specified in *duty_cycle_sp*. Valid values are *on* and *off*.

Speed_Regulation_P

int, read/write

The proportional constant for the speed regulation PID.

Speed_Regulation_I

int, read/write

The integral constant for the speed regulation PID.

Speed_Regulation_D

int, read/write

The derivative constant for the speed regulation PID.

State

string array, read

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

Stop_Command

string, read/write

Reading returns the current stop command. Writing sets the stop command. 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_commands* for a list of possible values.

Stop_Commands

string array, read

Returns a list of stop modes 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 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.

Time_SP

int, read/write

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.

1.1.3 Large Motor

class Large_Motor

EV3 large servo motor

inherits from: `motor`

Target driver(s): `lego-ev3-1-motor`

1.1.4 Medium Motor

class Medium_Motor

EV3 medium servo motor

inherits from: `motor`

Target driver(s): `lego-ev3-m-motor`

1.1.5 DC Motor

class `DC_Motor`

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.

ev3dev docs link: <http://www.ev3dev.org/docs/drivers/dc-motor-class/>

System properties

Address

string, read

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

Command

string, write

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

string array, read

Returns a list of commands supported by the motor controller.

Driver_Name

string, read

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

Duty_Cycle

int, read

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

Duty_Cycle_SP

int, read/write

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

string, read/write

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

Ramp_Down_SP

int, read/write

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

int, read/write

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.

State

string array, read

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_Command

string, write

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

Stop_Commands

string array, read

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

Time_SP

int, read/write

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.

1.1.6 Servo Motor

class Servo_Motor

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

ev3dev docs link: <http://www.ev3dev.org/docs/drivers/servo-motor-class/>

System properties**Address**

string, read

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

Command

string, write

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

string, read

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

Max_Pulse_SP

int, read/write

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

int, read/write

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

int, read/write

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

string, read/write

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

int, read/write

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

int, read/write

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.

State

string array, read

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

1.1.7 LED

class LED

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

System properties**Max_Brightness**

int, read

Returns the maximum allowable brightness value.

Brightness

int, read/write

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

Triggers

string array, read

Returns a list of available triggers.

Trigger

string selector, read/write

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.

Delay_On

int, read/write

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.

Delay_Off

int, read/write

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.

1.1.8 Button

class Button

Provides a generic button reading mechanism that can be adapted to platform specific implementations. Each platform's specific button capabilities are enumerated in the 'platforms' section of this specification.

1.1.9 Sensor

class Sensor

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.

ev3dev docs link: <http://www.ev3dev.org/docs/drivers/lego-sensor-class/>

System properties

Address

string, read

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*.

Command

string, write

Sends a command to the sensor.

Commands

string array, read

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

Decimals

int, read

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

Driver_Name

string, read

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

Mode

string, read/write

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

Modes

string array, read

Returns a list of the valid modes for the sensor.

Num_Values

int, read

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

Units

string, read

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

1.1.10 I2C Sensor

class I2C_Sensor

A generic interface to control I2C-type EV3 sensors.

inherits from: `sensor`

Target driver(s): `nxt-i2c-sensor`

System properties

FW_Version

string, read

Returns the firmware version of the sensor if available. Currently only I2C/NXT sensors support this.

Poll_MS

int, read/write

Returns the polling period of the sensor in milliseconds. Writing sets the polling period. Setting to 0 disables polling. Minimum value is hard coded as 50 msec. Returns -EOPNOTSUPP if changing polling is not supported. Currently only I2C/NXT sensors support changing the polling period.

1.1.11 Power Supply

class Power_Supply

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

System properties

Measured_Current

int, read

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

Measured_Voltage

int, read

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

Max_Voltage

int, read

Min_Voltage

int, read

Technology

string, read

Type

string, read

1.1.12 Lego Port

class Lego_Port

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.

System properties

Address

string, read

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

Driver_Name

string, read

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

Modes

string array, read

Returns a list of the available modes of the port.

Mode

string, read/write

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 and new ones loaded, however this will depend on the individual driver implementing this class.

Set_Device

string, write

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

string, read

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.

1.2 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.

1.2.1 Touch Sensor

class `Touch_Sensor`

Touch Sensor

inherits from: `sensor`

Target driver(s): `lego-ev3-touch`, `lego-nxt-touch`

Special properties

`Is_Pressed`

`boolean`, `read`

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

Required mode: `TOUCH`

Value index: 0

1.2.2 Color Sensor

class `Color_Sensor`

LEGO EV3 color sensor.

inherits from: `sensor`

Target driver(s): `lego-ev3-color`

ev3dev docs link: <http://www.ev3dev.org/docs/sensors/lego-ev3-color-sensor/>

Special properties

`Reflected_Light_Intensity`

`int`, `read`

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

Required mode: `COL-REFLECT`

Value index: 0

`Ambient_Light_Intensity`

`int`, `read`

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

Required mode: COL-AMBIENT

Value index: 0

Color

int, read

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

Required mode: COL-COLOR

Value index: 0

Red

int, read

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

Required mode: RGB-RAW

Value index: 0

Green

int, read

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

Required mode: RGB-RAW

Value index: 1

Blue

int, read

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

Required mode: RGB-RAW

Value index: 2

1.2.3 Ultrasonic Sensor

class Ultrasonic_Sensor

LEGO EV3 ultrasonic sensor.

inherits from: sensor

Target driver(s): lego-ev3-us, lego-nxt-us

ev3dev docs link: <http://www.ev3dev.org/docs/sensors/lego-ev3-ultrasonic-sensor/>

Special properties

Distance_Centimeters

float, read

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

Required mode: US-DIST-CM

Value index: 0

Distance_Inches

float, read

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

Required mode: US-DIST-IN

Value index: 0

Other_Sensor_Present

boolean, read

Value indicating whether another ultrasonic sensor could be heard nearby.

Required mode: US-LISTEN

Value index: 0

1.2.4 Gyro Sensor

class Gyro_Sensor

LEGO EV3 gyro sensor.

inherits from: `sensor`

Target driver(s): `lego-ev3-gyro`

ev3dev docs link: <http://www.ev3dev.org/docs/sensors/lego-ev3-gyro-sensor/>

Special properties

Angle

`int, read`

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

Required mode: `GYRO-ANG`

Value index: 0

Rate

`int, read`

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

Required mode: `GYRO-RATE`

Value index: 0

1.2.5 Infrared Sensor

class `Infrared_Sensor`

LEGO EV3 infrared sensor.

inherits from: `sensor`

Target driver(s): `lego-ev3-ir`

ev3dev docs link: <http://www.ev3dev.org/docs/sensors/lego-ev3-infrared-sensor/>

Special properties

Proximity

`int, read`

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

Required mode: IR-PROX

Value index: 0

1.2.6 Sound Sensor

class **Sound_Sensor**

LEGO NXT Sound Sensor

inherits from: `sensor`

Target driver(s): `lego-nxt-sound`

ev3dev docs link: <http://www.ev3dev.org/docs/sensors/lego-nxt-sound-sensor/>

Special properties

Sound_Pressure

`float, read`

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

Required mode: DB

Value index: 0

Sound_Pressure_Low

`float, read`

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

Required mode: DBA

Value index: 0

1.2.7 Light Sensor

class **Light_Sensor**

LEGO NXT Light Sensor

inherits from: `sensor`

Target driver(s): `lego-nxt-light`

ev3dev docs link: <http://www.ev3dev.org/docs/sensors/lego-nxt-light-sensor/>

Special properties

Reflected_Light_Intensity

`float`, `read`

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

Required mode: `REFLECT`

Value index: 0

Ambient_Light_Intensity

`float`, `read`

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

Required mode: `AMBIENT`

Value index: 0

1.3 Constants / Enums

Due to the inherent differences between the various languages that we support here, the enums listed below can also be declared as global constants.

1.3.1 Ports

INPUT_AUTO

Automatic input selection. Value is instance-specific (see below for details)

OUTPUT_AUTO

Automatic output selection. Value is instance-specific (see below for details)

INPUT_1

Sensor port 1, `"in1"`

INPUT_2

Sensor port 2,, `"in2"`

INPUT_3

Sensor port 3, `"in3"`

INPUT_4

Sensor port 4, `"in4"`

OUTPUT_A

Motor port A, `"outA"`

OUTPUT_B

Motor port B, "outB"

OUTPUT_C

Motor port C, "outC"

OUTPUT_D

Motor port D, "outD"

Note: The values for the *_AUTO constants can be chosen by the implementation. They can have any value that signifies an auto-search.

Compatibility table

Spec Version	Fully Supported Kernel Version
v0.9.1	v3.16.1-7-ev3dev
v0.9.2	v3.16.7-ckt10-4-ev3dev
v1.0.0	v3.16.7-ckt21-9-ev3dev

Autogen scripts for ev3dev language bindings

To help us maintain our language bindings, we have written a script to automatically update sections of code according to changes in our API specification. We define code templates using [Liquid](#), which are stored in the `templates` folder with the `.liquid` extension.

2.1 Usage

2.1.1 Prerequisites

- Make sure that you have cloned the repo (including submodules) and `cd'd` into the autogen directory
- You must have Node.JS and `npm` installed
- If you have not yet done so yet, run `npm install` to install the dependencies for auto-generation

2.1.2 Running from the command line

If you run the script without any parameters, it will look for an `autogen-config.json` file in your current working directory. The `autogen-config` file specifies the locations to look for templates and source files.

```
$ node path/to/autogen.js
```

If you want to specify a config file manually, you can include use a full file path for the target JSON config file.

```
$ node path/to/autogen.js other/path/to/config.json
```

2.2 How it works

Our script searches code files for comments that define blocks of code that it should automatically generate. The inline comment tags are formatted as follows (C-style comments):

```
//~autogen test-template foo.bar>tmp  
...  
//~autogen
```

After the initial declaration (`~autogen`), the rest of the comment defines parameters for the generation script. The first block of text up to the space is the file name of the template to use. The `.liquid` extension is automatically appended to the given name, and then the file is loaded and parsed.

The rest of the comment is a space-separated list of contextual variables. The section before the > defines the source, and the section after defines the destination. The value from the source is copied to the destination, in the global Liquid context. This makes it possible to use a single template file to generate multiple classes.

2.3 Implementation Notes

Todo

fill this up

2.4 Contributing to the autogen script

Just as we welcome contributions to the language bindings in this repo, we love to have people update our infrastructure. If you want to make a contribution, here are some quick tips to get started developing.

- After making a change, you should run the script and tell it to re-generate some files to make sure that your changes work as expected.
- If you are making more extensive changes, it may be helpful to create a temporary regen group with some test files to be able to manually test any new features or modifications.
- Although you can use any text editor, we recommend using [Visual Studio Code](#) to edit the autogen scripts. It has great autocomplete, and their debug GUI works well with node.

Indices and tables

- [genindex](#)
- [search](#)

A

Address (DC_Motor attribute), 8
 Address (Lego_Port attribute), 14
 Address (Motor attribute), 4
 Address (Sensor attribute), 12
 Address (Servo_Motor attribute), 9
 Ambient_Light_Intensity (Color_Sensor attribute), 15
 Ambient_Light_Intensity (Light_Sensor attribute), 20
 Angle (Gyro_Sensor attribute), 18

B

Blue (Color_Sensor attribute), 16
 Brightness (LED attribute), 11
 Button (built-in class), 11

C

Color (Color_Sensor attribute), 16
 Color_Sensor (built-in class), 15
 Command (DC_Motor attribute), 8
 Command (Motor attribute), 4
 Command (Sensor attribute), 12
 Command (Servo_Motor attribute), 9
 Commands (DC_Motor attribute), 8
 Commands (Motor attribute), 4
 Commands (Sensor attribute), 12
 connected (Device attribute), 4
 Count_Per_Rot (Motor attribute), 5

D

DC_Motor (built-in class), 8
 Decimals (Sensor attribute), 12
 Delay_Off (LED attribute), 11
 Delay_On (LED attribute), 11
 Device (built-in class), 4
 Distance_Centimeters (Ultrasonic_Sensor attribute), 17
 Distance_Inches (Ultrasonic_Sensor attribute), 17
 Driver_Name (DC_Motor attribute), 8
 Driver_Name (Lego_Port attribute), 14
 Driver_Name (Motor attribute), 5
 Driver_Name (Sensor attribute), 12

Driver_Name (Servo_Motor attribute), 9
 Duty_Cycle (DC_Motor attribute), 8
 Duty_Cycle (Motor attribute), 5
 Duty_Cycle_SP (DC_Motor attribute), 8
 Duty_Cycle_SP (Motor attribute), 5

E

Encoder_Polarity (Motor attribute), 5

F

FW_Version (I2C_Sensor attribute), 13

G

Green (Color_Sensor attribute), 16
 Gyro_Sensor (built-in class), 17

I

I2C_Sensor (built-in class), 12
 Infrared_Sensor (built-in class), 18
 INPUT_1 (built-in variable), 20
 INPUT_2 (built-in variable), 20
 INPUT_3 (built-in variable), 20
 INPUT_4 (built-in variable), 20
 INPUT_AUTO (built-in variable), 20
 Is_Pressed (Touch_Sensor attribute), 15

L

Large_Motor (built-in class), 7
 LED (built-in class), 10
 Lego_Port (built-in class), 13
 Light_Sensor (built-in class), 19

M

Max_Brightness (LED attribute), 10
 Max_Pulse_SP (Servo_Motor attribute), 9
 Max_Voltage (Power_Supply attribute), 13
 Measured_Current (Power_Supply attribute), 13
 Measured_Voltage (Power_Supply attribute), 13
 Medium_Motor (built-in class), 7
 Mid_Pulse_SP (Servo_Motor attribute), 10

Min_Pulse_SP (Servo_Motor attribute), 10
Min_Voltage (Power_Supply attribute), 13
Mode (Lego_Port attribute), 14
Mode (Sensor attribute), 12
Modes (Lego_Port attribute), 14
Modes (Sensor attribute), 12
Motor (built-in class), 4

N

Num_Values (Sensor attribute), 12

O

Other_Sensor_Present (Ultrasonic_Sensor attribute), 17
OUTPUT_A (built-in variable), 20
OUTPUT_AUTO (built-in variable), 20
OUTPUT_B (built-in variable), 20
OUTPUT_C (built-in variable), 21
OUTPUT_D (built-in variable), 21

P

Polarity (DC_Motor attribute), 8
Polarity (Motor attribute), 5
Polarity (Servo_Motor attribute), 10
Poll_MS (I2C_Sensor attribute), 13
Position (Motor attribute), 5
Position_D (Motor attribute), 6
Position_I (Motor attribute), 6
Position_P (Motor attribute), 5
Position_SP (Motor attribute), 6
Position_SP (Servo_Motor attribute), 10
Power_Supply (built-in class), 13
Proximity (Infrared_Sensor attribute), 18

R

Ramp_Down_SP (DC_Motor attribute), 8
Ramp_Down_SP (Motor attribute), 6
Ramp_Up_SP (DC_Motor attribute), 8
Ramp_Up_SP (Motor attribute), 6
Rate (Gyro_Sensor attribute), 18
Rate_SP (Servo_Motor attribute), 10
Red (Color_Sensor attribute), 16
Reflected_Light_Intensity (Color_Sensor attribute), 15
Reflected_Light_Intensity (Light_Sensor attribute), 20

S

Sensor (built-in class), 11
Servo_Motor (built-in class), 9
Set_Device (Lego_Port attribute), 14
Sound_Pressure (Sound_Sensor attribute), 19
Sound_Pressure_Low (Sound_Sensor attribute), 19
Sound_Sensor (built-in class), 19
Speed (Motor attribute), 6
Speed_Regulation_D (Motor attribute), 7

Speed_Regulation_Enabled (Motor attribute), 6
Speed_Regulation_I (Motor attribute), 6
Speed_Regulation_P (Motor attribute), 6
Speed_SP (Motor attribute), 6
State (DC_Motor attribute), 9
State (Motor attribute), 7
State (Servo_Motor attribute), 10
Status (Lego_Port attribute), 14
Stop_Command (DC_Motor attribute), 9
Stop_Command (Motor attribute), 7
Stop_Commands (DC_Motor attribute), 9
Stop_Commands (Motor attribute), 7

T

Technology (Power_Supply attribute), 13
Time_SP (DC_Motor attribute), 9
Time_SP (Motor attribute), 7
Touch_Sensor (built-in class), 15
Trigger (LED attribute), 11
Triggers (LED attribute), 11
Type (Power_Supply attribute), 13

U

Ultrasonic_Sensor (built-in class), 17
Units (Sensor attribute), 12