

---

# **Mindstorms EV3 Toolbox Documentation**

***Release v0.4-rc.10***

**LfB - RWTH Aachen**

**Dec 13, 2016**



## CONTENTS

<b>1</b>	<b>EV3</b>	<b>3</b>
<b>2</b>	<b>Motor</b>	<b>7</b>
<b>3</b>	<b>Sensor</b>	<b>11</b>
<b>4</b>	<b>Indices and tables</b>	<b>15</b>
	<b>MATLAB Module Index</b>	<b>17</b>
	<b>Index</b>	<b>19</b>



Contents:



## EV3

**class** `source.EV3` (*varargin*)

High-level class to work with physical bricks.

This is the ‘central’ class (from user’s view) when working with this toolbox. It delivers a convenient interface for creating a connection to the brick and sending commands to it. An EV3-object creates 4 Motor- and 4 Sensor-objects, one for each port.

### Notes

- Creating multiple EV3 objects and connecting them to different physical bricks has not been thoroughly tested yet, but seems to work on a first glance.

#### **motorA**

*Motor* – Motor-object interfacing port A

#### **motorB**

*Motor* – Motor-object interfacing port B

#### **motorC**

*Motor* – Motor-object interfacing port C

#### **motorD**

*Motor* – Motor-object interfacing port D

#### **sensor1**

*Sensor* – Motor-object interfacing port 1

#### **sensor2**

*Sensor* – Motor-object interfacing port 2

#### **sensor3**

*Sensor* – Motor-object interfacing port 3

#### **sensor4**

*Sensor* – Motor-object interfacing port 4

#### **debug**

*numeric in {0,1,2}* – Debug mode. [*WRITABLE*]

- 0: Debug turned off
- 1: Debug turned on for EV3-object -> enables feedback in the console about what firmware-commands have been called when using a method
- 2: Low-level-Debug turned on -> each packet sent and received is printed to the console

**batteryMode**

*string in {'Percentage', 'Voltage'}* – Mode for reading battery charge. *[WRITABLE]*

**batteryValue**

*numeric* – Current battery charge. Depending on batteryMode, the reading is either in percentage or voltage. *[READ-ONLY]*

**isConnected**

*bool* – True if virtual brick-object is connected to physical one. *[READ-ONLY]*

## Examples

```
b = EV3(); b.connect('usb'); ma = b.motorA; ma.setProperties('power', 50, 'limitValue', 720); ma.start(); %  
fun b.sensor1.value b.waitFor(); b.beep(); delete b;
```

**beep** (*ev3*)

Plays a 'beep'-tone on brick.

## Notes

- This equals playTone(10, 1000, 100) (Wraps the same opCode in comm-layer)

## Example

```
b = EV3(); b.connect('bt', 'serPort', '/dev/rfcomm0'); b.beep();
```

**connect** (*ev3*, *varargin*)

Connects EV3-object and its Motors and Sensors to physical brick.

**Parameters**

- **connectionType** (*string in {'bt', 'usb'}*) – Connection type
- **serPort** (*string in {'/dev/rfcomm1', '/dev/rfcomm2', ...}*) – Path to serial port (if 'bt')
- **beep** (*bool*) – If true, EV3 beeps if connection has been established

## Examples

```
% Setup bluetooth connection via com-port 0 b = EV3(); b.connect('bt', 'serPort', '/dev/rfcomm0'); %  
Setup usb connection, beep when connection has been established b = EV3(); b.connect('usb', 'beep',  
'on', );
```

Check connection

**disconnect** (*ev3*)

Disconnects EV3-object and its Motors and Sensors from physical brick.

## Notes

- Gets called automatically when EV3-object is destroyed.



### Example

```
b = EV3(); b.connect('bt', 'serPort', '/dev/rfcomm0'); % do stuff b.disconnect();
```

Reset motors and sensors before disconnecting

**playTone** (*ev3*, *volume*, *frequency*, *duration*)

Plays tone on brick.

#### Parameters

- **volume** (*numeric in [0, 100]*) – in percent
- **frequency** (*numeric in [250, 10000]*) – in Hertz
- **duration** (*numeric >0*) – in milliseconds

### Example

```
b = EV3(); b.connect('bt', 'serPort', '/dev/rfcomm0'); b.playTone(40, 5000, 1000); % Plays tone with 40% volume and 5000Hz for 1 second.
```

**setProperty**s (*ev3*, *varargin*)

Set multiple EV3 properties at once using MATLAB's inputParser.

#### Parameters

- **debug** (*numeric in {0,1,2}*) – see EV3.debug [OPTIONAL]
- **batteryMode** (*string in {'Voltage'/'Percentage'}*) – see EV3.batteryMode [OPTIONAL]

### Example

```
b = EV3(); b.connect('bt', 'serPort', '/dev/rfcomm0'); b.setProperty('debug', 'on', 'batteryMode', 'Voltage'); % Instead of: b.debug = 'on'; b.batteryMode = 'Voltage';
```

See also EV3.DEBUG, EV3.BATTERYMODE

**stopAllMotors** (*ev3*)

Sends a stop-command to all motor-ports

**stopTone** (*ev3*)

Stops tone currently played

### Example

```
b = EV3(); b.connect('bt', 'serPort', '/dev/rfcomm0'); b.playTone(10,100,1000000000); % Accidentally given wrong tone duration :) b.stopTone(); % Stops tone immediately.
```

**tonePlayed** (*ev3*)

Tests if tone is currently played.

**Returns** *status* – True if a tone is being played

**Return type** bool

**Example** `b = EV3(); b.connect('bt', 'serPort', '/dev/rfcomm0'); b.playTone(10, 100, 1000); pause(0.5); b.tonePlayed() -> Outputs 1 to console.`



## MOTOR

**class** `source.Motor` (*varargin*)

High-level class to work with motors.

This class is supposed to ease the use of the brick's motors. It is possible to set all kinds of parameters, request the current status of the motor ports and of course send commands to the brick to be executed on the respective port.

### Notes

- You don't need to create instances of this class. The EV3-class automatically creates instances for each motor port, and you can work with them via the EV3-object.
- The Motor-class represents motor ports, not individual motors!
- If you start a motor with power=0, the internal state will still be set to 'isRunning'

### **power**

*numeric in [-100, 100]* – Power level of motor in percent. [WRITABLE]

### **speedRegulation**

*bool* – Speed regulation turned on or off. When turned on, motor will try to 'hold' its speed at given power level, whatever the load. In this mode, the highest possible speed depends on the load and mostly goes up to around 70-80 (at this point, the Brick internally input 100% power). When turned off, motor will constantly input the same power into the motor. The resulting speed will be somewhat lower, depending on the load. [WRITABLE]

### **smoothStart**

*numeric s. t. smoothStart+smoothStop < limitValue* – Degrees/Time indicating how far/long the motor should smoothly start. Depending on limitMode, the input is interpreted either in degrees or milliseconds. The first {smoothStart}-milliseconds/degrees of limitValue the motor will slowly accelerate until reaching its defined speed. [WRITABLE]

### **smoothStop**

*numeric s. t. smoothStart+smoothStop < limitValue* – Degrees/Time indicating how far/long the motor should smoothly stop. Depending on limitMode, the input is interpreted either in degrees or milliseconds. The last [smoothStop]-milliseconds/degrees of limitValue the motor will slowly slow down until it has stopped. [WRITABLE]

### **limitValue**

*numeric >=0* – Degrees/Time indicating how far/long the motor should run. Depending on limitMode, the input is interpreted either in degrees or milliseconds. [WRITABLE]

### **limitMode**

'Tacho'|'Time' – Mode for motor limit. [WRITABLE]

**brakeMode**

'Brake'/'Coast' – Action done when stopping. If 'Coast', the motor will (at tacholimit, if  $\sim=0$ ) coast to a stop. If 'Brake', the motor will stop immediately (at tacholimit, if  $\sim=0$ ) and hold the brake. [WRITABLE]

**debug**

*bool* – Debug turned on or off. In debug mode, everytime a command is passed to the sublayer ('communication layer'), there is feedback in the console about what command has been called. [WRITABLE]

**isRunning**

*bool* – True if motor is running. [READ-ONLY]

**tachoCount**

*numeric* – Current tachometer count. [READ-ONLY]

**currentSpeed**

*numeric* – Current speed of motor. If speedRegulation=on this should equal power, otherwise it will probably be lower than that. [READ-ONLY]

**type**

*DeviceType* – Type of connected device if any. [READ-ONLY]

**internalReset** (*motor*)

Resets internal tachometer count. Use this if motor behaves weird (i.e. not starting at all, or not correctly running to limitValue)

The internal tachometer count is used for positioning the motor. When the motor is running with a tachometer limit, internally it uses another counter than the one read by tachometerCount. This internal tachometer count needs to be reset if you physically change the motor's position or it coasted into a stop. If the motor's brakemode is 'Coast', this function is called automatically.

**Notes**

- A better name would probably be resetPosition...
- Gets called automatically when starting the motor and the internal tachometer

count is  $> 0$

See also MOTOR.RESETTACHOCOUNT

**resetTachoCount** (*motor*)

Resets tachometer count

**setBrake** (*motor*, *brake*)

Apply or release brake of motor

**Parameters** *brake* (*bool*) – If true, brake will be pulled

**setProperty** (*motor*, *varargin*)

Sets multiple Motor properties at once using MATLAB's inputParser.

**Parameters**

- **debug** (*bool*) –
- **smoothStart** (*numeric* in  $[0, \text{limitValue}]$ ) –
- **smoothStop** (*numeric* in  $[0, \text{limitValue}]$ ) –
- **speedRegulation** (*bool*) –
- **brakeMode** ('Coast'/'Brake') –

- **limitMode** (*'Time' | 'Tacho'*) –
- **limitValue** (*numeric > 0*) –
- **power** (*numeric in [-100, 100]*) –
- **batteryMode** (*'Voltage' | 'Percentage'*) –

### Example

```
b = EV3(); b.connect('bt', 'serPort', '/dev/rfcomm0'); b.motorA.setProperties('debug', 'on', 'power', 50, 'limitValue', 720, 'speedRegulation', 'on'); % Instead of: b.motorA.debug = 'on'; % b.motorA.power = 50; % b.motorA.limitValue = 720; % b.motorA.speedRegulation = 'on';
```

#### **start** (*motor*)

Starts the motor

### Notes

- Right now, alternatingly calling this function with and without tacho limit may lead to unexpected behaviour. For example, if you run the motor without a tacholimit for some time using Coast, then stop using Coast, and then try to run the with a tacholimit, it will stop sooner or later than expected, or may not even start at all.
- (OLD) After calling one of the functions to control the motor with some kind of limit (which is done if `limit~=0`), the physical brick's power/speed value for starting without a limit (i.e. if `limit==0`) is reset to zero. So if you want to control the motor without a limit after doing so with a limit, you would have to set the power manually to the desired value again. (I don't really know if this is deliberate or a bug, and at this point, I'm too afraid to ask.) To avoid confusion, this is done automatically in this special case. However, this does not even work all the time. If motor does not start, call `stop()` and `setPower()` manually. ./

Check connection and if motor is already running

#### **stop** (*motor*)

Stops the motor

#### **syncedStart** (*motor, syncMotor, varargin*)

Starts this motor synchronized with another

This motor acts as a 'master', meaning that the synchronized control is done via this one. When `syncedStart` is called, the master sets some of the slave's (`syncMotor`) properties to keep it consistent with the physical brick. So, for example, changing the power on the master motor will take effect on the slave as soon as this method is called. The following parameters will be affected on the slave: `power`, `brakeMode`, `limitValue`, `speedRegulation`

#### Parameters

- **syncMotor** (*Motor*) – the motor-object to sync with
- **turnRatio** (*numeric in [-200, 200]*) – [OPTIONAL] (Excerpt of Firmware-comments, in `c_output.c`): "Turn ratio is how tight you turn and to what direction you turn.
  - 0 value is moving straight forward
  - Negative values turn to the left
  - Positive values turn to the right

- Value -100 stops the left motor
- Value +100 stops the right motor
- Values less than -100 makes the left motor run the opposite direction of the right motor (Spin)
- Values greater than +100 makes the right motor run the opposite direction of the left motor (Spin)”

### Notes

- This is right now a pretty ‘heavy’ function, as it tests if both motors are connected AND aren’t running, wasting four packets, keep that in mind
- It is necessary to call `syncdStop()` and not `stop()` for stopping the motors (otherwise the sync-state cannot be exited correctly)

### Example

```
b = EV3(); b.connect('usb'); m = b.motorA; slave = b.motorB; m.power = 50; m.syncdStart(slave); % Do stuff m.syncdStop();
```

#### **syncdStop** (*motor*)

Stops both motors previously started with `syncdStart`.

See also `MOTOR.SYNCEDSTART`

#### **waitFor** (*motor*)

Stops execution of program as long as motor is running

### Notes

- (OLD)This one’s a bit tricky. The opCode `OutputReady` makes the brick stop sending responses until the motor has stopped. For security reasons, in this toolbox there is an internal timeout for receiving messages from the brick. It raises an error if a reply takes too long, which would happen in this case. As a workaround, there is an infinite loop that catches errors from `outputReady` and continues then, until `outputReady` will actually finish without an error.
- (OLD)`OutputReady` (like `OutputTest` in `isRunning`) sometimes doesn’t work. If `outputReady` returns in less than a second, another while-loop iterates until the motor has stopped, this time using `motor.isRunning()` (this only works as long as not both `OutputTest` and `OutputReady` are buggy).
- (OLD)Workaround: Poll `isRunning` (which itself return `(speed>0)`) until it is false (No need to check if motor is connected as `speed` correctly returns 0 if it’s not)

## SENSOR

**class** `source.Sensor` (*varargin*)

High-level class to work with sensors.

The Sensor-class facilitates the communication with sensors. This mainly consists of reading the sensor's type and current value in a specified mode.

### Notes

- You don't need to create instances of this class. The EV3-class automatically creates instances for each sensor port, and you can work with them via the EV3-object.
- The Sensor-class represents sensor ports, not individual sensors!

### mode

*DeviceMode.{Type}* – Sensor mode in which the value will be read. By default, mode is set to *DeviceMode.Default.Undefined*. Once a physical sensor is connected to the port *and* the physical Brick is connected to the EV3-object, the allowed mode and the default mode for a Sensor-object are the following (depending on the sensor type): *[WRITABLE]*

#### •Touch-Sensor:

- *DeviceMode.Touch.Pushed [Default]*
- *DeviceMode.Touch.Bumps*

#### •Ultrasonic-Sensor:

- *DeviceMode.UltraSonic.DistCM [Default]*
- *DeviceMode.UltraSonic.DistIn*
- *DeviceMode.UltraSonic.Listen*

#### •Color-Sensor:

- *DeviceMode.Color.Reflect [Default]*
- *DeviceMode.Color.Ambient*
- *DeviceMode.Color.Col*

#### •Gyro-Sensor:

- *DeviceMode.Gyro.Angular [Default]*
- *DeviceMode.Gyro.Rate*

#### •Infrared-Sensor:

- DeviceMode.InfraRed.Prox *[Default]*
- DeviceMode.InfraRed.Seek
- DeviceMode.InfraRed.Remote

•**NXTCOLOR-SENSOR:**

- DeviceMode.NXTCOLOR.Reflect *[Default]*
- DeviceMode.NXTCOLOR.Ambient
- DeviceMode.NXTCOLOR.Color
- DeviceMode.NXTCOLOR.Green
- DeviceMode.NXTCOLOR.Blue
- DeviceMode.NXTCOLOR.Raw

•**NXTLIGHT-SENSOR:**

- DeviceMode.NXTLIGHT.Reflect *[Default]*
- DeviceMode.NXTLIGHT.Ambient

•**NXTSOUND-SENSOR:**

- DeviceMode.NXTSOUND.DB *[Default]*
- DeviceMode.NXTSOUND.DBA

•**NXTTEMPERATURE-SENSOR**

- DeviceMode.NXTTEMPERATURE.C *[Default]*
- DeviceMode.NXTTEMPERATURE.F

•**NXTTOUCH-SENSOR:**

- DeviceMode.NXTTOUCH.Pushed *[Default]*
- DeviceMode.NXTTOUCH.Bumps

•**NXTULTRASONIC-SENSOR:**

- DeviceMode.NXTULTRASONIC.CM *[Default]*
- DeviceMode.NXTULTRASONIC.IN

•**HTACCELEROMETER-SENSOR:**

- DeviceMode.HTACCELEROMETER.Acceleration *[Default]*
- DeviceMode.HTACCELEROMETER.AccelerationAllAxes

•**HTCOMPASS-SENSOR:**

- DeviceMode.HTCOMPASS.Degrees *[Default]*

•**HTCOLOR-SENSOR:**

- DeviceMode.HTCOLOR.Col *[Default]*
- DeviceMode.HTCOLOR.Red
- DeviceMode.HTCOLOR.Green
- DeviceMode.HTCOLOR.Blue
- DeviceMode.HTCOLOR.White



- DeviceMode.HTColor.Raw
- DeviceMode.HTColor.Nr,
- DeviceMode.HTColor.All

**debug**

*bool* – Debug turned on or off. In debug mode, everytime a command is passed to the sublayer ('communication layer'), there is feedback in the console about what command has been called. [WRITABLE]

**value**

*numeric* – Value read from hysical sensor. What the value represents depends on sensor.mode. [READ-ONLY]

**type**

*DeviceType* – Type of physical sensor connected to the port. Possible types are: [READ-ONLY]

- DeviceType.NXTTouch
- DeviceType.NXTLight
- DeviceType.NXTSound
- DeviceType.NXTColor
- DeviceType.NXTUltraSonic
- DeviceType.NXTTemperature
- DeviceType.LargeMotor
- DeviceType.MediumMotor
- DeviceType.Touch
- DeviceType.Color
- DeviceType.UltraSonic
- DeviceType.Gyro
- DeviceType.InfraRed
- DeviceType.HTColor
- DeviceType.HTCompass
- DeviceType.HTAccelerometer
- DeviceType.Unknown
- DeviceType.None
- DeviceType.Error

**reset** (*sensor*)

Resets value on sensor

**Notes**

- This clears ALL the sensors right now, no other Op-Code available... :(

**setProperty**s (*sensor*, *varargin*)

Sets multiple Sensor properties at once using MATLAB's inputParser.

**Parameters**

- **debug** (*bool*) –
- **mode** (*DeviceMode.Type*) –

### Example

```
b = EV3(); b.connect('bt', 'serPort', '/dev/rfcomm0'); b.sensor1.setProperties('debug', 'on', 'mode',  
DeviceMode.Color.Ambient); % Instead of: b.sensor1.debug = 'on'; % b.sensor1.mode = Device-  
Mode.Color.Ambient;
```

## INDICES AND TABLES

- *genindex*
- *modindex*
- *search*



**S**

source, [1](#)



## B

batteryMode (source.EV3 attribute), 3  
 batteryValue (source.EV3 attribute), 4  
 beep() (source.EV3 method), 4  
 brakeMode (source.Motor attribute), 7

## C

connect() (source.EV3 method), 4  
 currentSpeed (source.Motor attribute), 8

## D

debug (source.EV3 attribute), 3  
 debug (source.Motor attribute), 8  
 debug (source.Sensor attribute), 13  
 disconnect() (source.EV3 method), 4

## E

EV3 (class in source), 3

## I

internalReset() (source.Motor method), 8  
 isConnected (source.EV3 attribute), 4  
 isRunning (source.Motor attribute), 8

## L

limitMode (source.Motor attribute), 7  
 limitValue (source.Motor attribute), 7

## M

mode (source.Sensor attribute), 11  
 Motor (class in source), 7  
 motorA (source.EV3 attribute), 3  
 motorB (source.EV3 attribute), 3  
 motorC (source.EV3 attribute), 3  
 motorD (source.EV3 attribute), 3

## P

playTone() (source.EV3 method), 5  
 power (source.Motor attribute), 7

## R

reset() (source.Sensor method), 13

resetTachoCount() (source.Motor method), 8

## S

Sensor (class in source), 11  
 sensor1 (source.EV3 attribute), 3  
 sensor2 (source.EV3 attribute), 3  
 sensor3 (source.EV3 attribute), 3  
 sensor4 (source.EV3 attribute), 3  
 setBrake() (source.Motor method), 8  
 setProperties() (source.EV3 method), 5  
 setProperties() (source.Motor method), 8  
 setProperties() (source.Sensor method), 13  
 smoothStart (source.Motor attribute), 7  
 smoothStop (source.Motor attribute), 7  
 source (module), 1  
 speedRegulation (source.Motor attribute), 7  
 start() (source.Motor method), 9  
 stop() (source.Motor method), 9  
 stopAllMotors() (source.EV3 method), 5  
 stopTone() (source.EV3 method), 5  
 syncedStart() (source.Motor method), 9  
 syncedStop() (source.Motor method), 10

## T

tachoCount (source.Motor attribute), 8  
 tonePlayed() (source.EV3 method), 5  
 type (source.Motor attribute), 8  
 type (source.Sensor attribute), 13

## V

value (source.Sensor attribute), 13

## W

waitFor() (source.Motor method), 10