

---

# **Mindstorms EV3 Toolbox Documentation**

*Release v1.0*

**LfB - RWTH Aachen**

**Aug 15, 2017**



## CONTENTS

<b>1</b>	<b>Contents</b>	<b>3</b>
1.1	EV3 . . . . .	3
1.2	Motor . . . . .	6
1.3	Sensor . . . . .	10
1.4	hidapi . . . . .	13
1.5	usbBrickIO . . . . .	17
1.6	btBrickIO . . . . .	18
	<b>MATLAB Module Index</b>	<b>21</b>
	<b>Index</b>	<b>23</b>



Hi there! This is the documentation for the “Lego Mindstorms EV3” MATLAB Toolbox, developed by RWTH Aachen. For an introduction about this toolbox, installation guides and examples, take a look at [our repository](#).



## CONTENTS

High-Level documentation

## 1.1 EV3

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

*List of methods:*

- `connect()`
- `disconnect()`
- `stopAllMotors()`
- `beep()`
- `playTone()`
- `stopTone()`
- `tonePlayed()`
- `setProperties()`

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.
- When an input argument of a method is marked as optional, the argument needs to be ‘announced’ by a preceding 2nd argument, which is a string containing the name of the argument. For example, `Motor.setProperties` may be given a power-parameter. The syntax would be as follows: *brickObject.motorA.setProperties('power', 50);*

#### **motorA**

*Motor* – Motor-object interfacing port A. See also *Motor*.

#### **motorB**

*Motor* – Motor-object interfacing port B. See also *Motor*.

**motorC**

*Motor* – Motor-object interfacing port C. See also *Motor*.

**motorD**

*Motor* – Motor-object interfacing port D. See also *Motor*.

**sensor1**

*Sensor* – Motor-object interfacing port 1. See also *Sensor*.

**sensor2**

*Sensor* – Motor-object interfacing port 2. See also *Sensor*.

**sensor3**

*Sensor* – Motor-object interfacing port 3. See also *Sensor*.

**sensor4**

*Sensor* – Motor-object interfacing port 4. See also *Sensor*.

**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. See also *batteryValue*. *[WRITABLE]*

**batteryValue**

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

**isConnected**

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

## Example

```
% This example expects a motor at port A and a (random) sensor at port 1
b = EV3(); % b.connect('usb'); %
ma = b.motorA; % ma.setProperties('power', 50, 'limitValue', 720); % ma.start(); % % fun ma.waitFor(); %
disp(b.sensor1.value); % b.beep(); % delete b; %
```

**beep** (*ev3*)

Plays a 'beep'-tone on brick.

## Notes

- This equals `playTone(10, 1000, 100)`.

## 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 (necessary if connectionType is 'bt'). [OPTIONAL]
- **beep** (*bool*) – If true, EV3 beeps if connection has been established. [OPTIONAL]

### Example

```
% 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', ); %
```

See also ISCONNECTED / isConnected

### 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(); %
```

Resetting needs a working connection in order to send reset-commands to the Brick. If the connection has been aborted (e.g. by pulling the USB-cord), the reset-methods would fail -> catch this error and for now do nothing.

### 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 (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.setProperties('debug', 'on', 'batteryMode',  
'Voltage'); % % Instead of: b.debug = 'on'; b.batteryMode = 'Voltage'; %
```

See also EV3.DEBUG, EV3.BATTERYMODE / debug, batteryMode

#### **stopTone** (ev3)

Stops tone currently played.

### Example

```
b = EV3(); % b.connect('bt', 'serPort', '/dev/rfcomm0'); % b.playTone(10,100,1000000000); % Acciden-  
tally 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); % Small pause is necessary as tone does not start instantaneously b.tonePlayed(); % -  
> Outputs 1 to console.
```

## 1.2 Motor

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

#### *List of methods:*

- `start()`
- `stop()`
- `syncedStart()`
- `syncedStop()`
- `waitFor()`
- `internalReset()`
- `resetTachoCount()`
- `setBrake()`
- `setProperties()`

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'
- When an input argument of a method is marked as optional, the argument needs to be 'announced' by a preceding 2nd argument, which is a string containing the name of the argument. For example, Motor.setProperties may be given a power-parameter. The syntax would be as follows: *brickObject.motorA.setProperties('power', 50);*

### 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 inputs 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. See also *limitValue*, *limitMode*. [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. See also *limitValue*, *limitMode*. [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. See also *limitMode*. [WRITABLE]

### limitMode

'Tacho'|'Time' – Mode for motor limit. See also *limitValue*. [WRITABLE]

### brakeMode

'Brake'|'Coast' – Action done when stopping. If 'Coast', the motor will (at tacholimit, if ~=0) coast to a stop. If 'Brake', the motor will stop immediately (at tacholimit, if ~=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 tacho count in degrees. [READ-ONLY]

### currentSpeed

*numeric* – Current speed of motor. If speedRegulation=on this should equal power, otherwise it will probably be lower than that. See also *speedRegulation*. [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 *tachoCount*. 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*

**resetTachoCount** (*motor*)

Resets tachometer count.

See also `MOTOR.TACHOCOUNT` / *tachoCount*

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

Apply or release brake of motor.

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

**Notes**

- This method does not affect `Motor.brakeMode`. After the next run, the motor will again be stopped as specified in `Motor.brakeMode`.

See also `MOTOR.BRAKEMODE` / *brakeMode*

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

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

**Parameters**

- **debug** (*bool*) – *[OPTIONAL]*
- **smoothStart** (*numeric in [0, limitValue]*) – *[OPTIONAL]*
- **smoothStop** (*numeric in [0, limitValue]*) – *[OPTIONAL]*
- **speedRegulation** (*bool*) – *[OPTIONAL]*
- **brakeMode** ('Coast' / 'Brake') – *[OPTIONAL]*
- **limitMode** ('Time' / 'Tacho') – *[OPTIONAL]*
- **limitValue** (*numeric > 0*) – *[OPTIONAL]*
- **power** (*numeric in [-100, 100]*) – *[OPTIONAL]*
- **batteryMode** ('Voltage' / 'Percentage') – *[OPTIONAL]*

## 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.

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

Stops the motor.

## Notes

- If this motor has been started synced with another one (either as master or slave, using `Motor.syncedStart`), `syncedStop()` will be called, stopping both motors.

See also `MOTOR.START`, `MOTOR.SYNCEDEDSTOP` / `start()`, `syncedStop()`

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

Starts this motor synchronized with another.

The motor, with which this method is called, acts as a *master*, meaning that the synchronized control is done with it and uses its parameters. 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, if the master has another power-value than the slave, the slave's power-value will be set to that of the master when `syncedStart()` 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]*) – Ratio between the two master's and the slave's motor speed. With values != 0 one motor will be slower than the other or even turn into the other direction. This can be used for turning car-like robots, for example. [OPTIONAL] (Read in Firmware-comments in `c_output.c`): -> 0 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 a pretty 'heavy' function, as it tests if both motors are connected AND aren't running, wasting four packets, keep that in mind.

## Example

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

See also `MOTOR.STOP`, `MOTOR.SYNCEDEDSTOP` / `stop()`, `syncedStop()`

**syncedStop** (*motor*)

Stops both motors previously started with syncedStart.

**Notes**

- This method is called automatically by stop(), if the motors have been started using syncedStart, and the regular stop-method has been called afterwards.

See also MOTOR.SYNCEDEDSTART, MOTOR.STOP / *syncedStart()*, *stop()*

**waitFor** (*motor*)

Stops execution of program as long as motor is running.

**Notes**

- This one's a bit tricky. The opCode which is supposed to be used here, 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.
- Workaround: Poll isRunning until it is false (No need to check if motor is connected as speed correctly returns 0 if it's not)

## 1.3 Sensor

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

*List of methods:*

- *reset()*
- *setProperties()*

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!
- When an input argument of a method is marked as optional, the argument needs to be 'announced' by a preceding 2nd argument, which is a string containing the name of the argument. For example, Motor.setProperties may be given a power-parameter. The syntax would be as follows: *brickObject.motorA.setProperties('power', 50);*

**mode**

*DeviceMode.{Type}* – Sensor mode in which the value will be read. By default, mode is set to *DeviceMode.Default.Undefined*. See also *type*. [WRITABLE] 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):

**•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 *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 sensor value.

### Notes

- Has not been thoroughly tested but seems to work as expected

**setPropertyies** (*sensor*, *varargin*)

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

#### Parameters

- **debug** (*bool*) – [OPTIONAL]
- **mode** (*DeviceMode.Type*) – [OPTIONAL]

### 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;
```

Low-Level documentation

## 1.4 hidapi

**class** `source.hidapi` (*vendorID*, *productID*, *nReadBuffer*, *nWriteBuffer*)

#### List of methods:

- `open()`
- `close()`
- `read()`
- `read_timeout()`
- `write()`

- `getHIDInfoString()`
- `setNonBlocking()`
- `init()`
- `exit()`
- `error()`
- `enumerate()`
- `getManufacturersString()`
- `getProductString()`
- `getSerialNumberString()`

Interface to the hidapi library

## Notes

- Developed from the hidapi available at <http://www.signal11.us/oss/hidapi/>.
- Windows: hidapi.dll needed.
- Mac: hidapi.dylib needed. In addition, Xcode has to be installed.
- Linux: hidapi has to be compiled on host-system.

**handle**

**vendorID**

*numeric* – Vendor-ID of the USB device.

**productID**

*numeric* – Product-ID of the USB device.

**nReadBuffer**

*numeric* – Read-buffer size in bytes.

**nWriteBuffer**

*numeric* – Write-buffer size in bytes. Needs to be 1 Byte bigger than actual packet.

**slib**

*string* – Name of shared library file (without file extension). Defaults to 'hidapi'.

**sheader**

*string* – Name of shared library header. Defaults to 'hidapi.h'.

## Example

```
hidHandle = hidapi(1684,0005,1024,1025); %\n
```

**close** (*hid*)

Close the connection to a hid device.

**Throws:** InvalidHandle: Handle to USB-device not valid

## Notes

- Gets called automatically when deleting the hidapi instance.

2001.Check if pointer is (unexpectedly) already invalidated

**enumerate** (*hid*, *vendorID*, *productID*)

Enumerates the info about the hid device with the given vendorID and productID and returns a string with the returned hid information.

### Parameters

- **vendorID** (*numeric*) – Vendor-ID of the USB device in decimal.
- **productID** (*numeric*) – Product-ID of the USB device in decimal.

## Notes

- Using a vendorID and productID of (0,0) will enumerate all connected hid devices.
- MATLAB does not have the hid\_device\_infoPtr struct so some of the returned information will need to be resized and cast into uint8 or chars.

Enumerate the hid devices

**error** (*hid*)

Return the hid device error string if a function produced an error.

**Throws:** InvalidHandle: Handle to USB-device not valid

## Notes

- This function must be called explicitly if you think an error was generated from the hid device.

2001.Check if pointer is (unexpectedly) already invalidated

**exit** (*hid*)

hidapi.exit Exit hidapi

hid.exit() exits the hidapi library.

**Throws:** CommError: Error during communication with device

Notes:: - You should not have to call this function directly.

**getHIDInfoString** (*hid*, *info*)

Get the corresponding hid info from the hid device.

**Throws:** CommError: Error during communication with device InvalidHandle: Handle to USB-device not valid

## Notes

- Info is the hid information string.

See also `HIDAPI.GETMANUFACTURERSSTRING`, `HIDAPI.GETPRODUCTSTRING`, `HIDAPI.GETSERIALNUMBERSTRING`.

Read buffer `nReadBuffer` length

**getManufacturersString** (*hid*)

Get manufacturers string from `hid` object using `getHIDInfoString`.

**getProductString** (*hid*)

Get product string from `hid` object using `getProductString`.

**getSerialNumberString** (*hid*)

Get serial number from `hid` object using `getSerialNumberString`.

**init** (*hid*)

Initiates the `hidapi` library.

**Throws:** `CommError`: Error during communication with device

### Notes

- This is called automatically in the library itself with the `open` function. You should not have to call this function directly.

**open** (*hid*)

Open a connection with a `hid` device

**Throws:** `CommError`: Error during communication with device

### Notes

- Gets called automatically when creating an `hidapi`-object.
- The pointer return value from this library call is always null so it is not possible to know if the open was successful.
- The final parameter to the open `hidapi` library call has different types depending on OS. On windows it is `uint16`, on linux/mac `int32`.

Create a null pointer for the `hid_open` function (depends on OS)

**read** (*hid*)

Read from a `hid` device and returns the read bytes.

**Throws:** `CommError`: Error during communication with device `InvalidHandle`: Handle to USB-device not valid

### Notes

- Will print an error if no data was read.

Read buffer of `nReadBuffer` length

**read\_timeout** (*hid, timeOut*)

Read from a `hid` device with a timeout and return the read bytes.

**Parameters** `timeOut` (*numeric*  $\geq 0$ ) – Milliseconds after which a timeout-error occurs if no packet could be read.

**Throws:** `CommError`: Error during communication with device `InvalidHandle`: Handle to USB-device not valid

Read buffer of `nReadBuffer` length

**setNonBlocking** (*hid*, *nonblock*)

Set the non blocking flag on the hid device connection.

**Parameters** `nonblock` (*numeric* in  $\{0, 1\}$ ) – 0 disables nonblocking, 1 enables non-blocking

**Throws:** `CommError`: Error during communication with device `InvalidHandle`: Handle to USB-device not valid

2001. Check if pointer is (unexpectedly) already invalidated

**write** (*hid*, *wmsg*, *reportID*)

Write to a hid device.

**Throws:** `CommError`: Error during communication with device `InvalidHandle`: Handle to USB-device not valid

### Notes

- Will print an error if there is a mismatch between the buffer size and the reported number of bytes written.

Append a 0 at the front for HID report ID

## 1.5 usbBrickIO

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

*List of methods:*

- `open()`
- `close()`
- `read()`
- `write()`
- `setProperties()`

USB interface between MATLAB and the brick

### Notes

- Uses the hid library implementation in `hidapi.m`
- The default parameters should always work when you try to connect to an EV3 brick, so in nearly all use-cases, the constructor does not need any parameters (besides 'debug' eventually).

**debug**

*bool* – If true, each open/close/read/write-call will be noted in the console. Defaults to false.

**vendorID**

*numeric* – Vendor-ID of the USB device. Defaults to 0x694 (EV3 vendor ID).

**productID**

*numeric* – Product-ID of the USB device. Defaults to 0x0005 (EV3 product ID).

**nReadBuffer**

*numeric* – Read-buffer size in bytes. Defaults to 1024.

**nWriteBuffer**

*numeric* – Write-buffer size in bytes. Needs to be 1 Byte bigger than actual packet. Defaults to 1025 (EV3 USB maximum packet size = 1024).

**timeOut**

*numeric*  $\geq 0$  – Milliseconds after which a timeout-error occurs if no packet could be read. Defaults to 10000.

## Examples

```
% Connecting via USB commHandle = usbBrickIO(); % % Connecting via USB with enabled debug output  
commHandle = usbBrickIO('debug', true); %
```

**close** (*brickIO*)

Closes the usb connection the brick through the hidapi interface.

**open** (*brickIO*)

Opens the usb connection to the brick through the hidapi interface.

**read** (*brickIO*)

Reads data from the brick through usb using the hidapi interface and returns the data in uint8 format.

**setProperty**s (*brickIO*, *varargin*)

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

The syntax is as follows: `commHandle.setProperty('propertyName1', propertyValue1, 'propertyName2', propertyValue2, ...)`. Valid, optional properties are: `debug`, `vendorID`, `productID`, `nReadBuffer`, `nWriteBuffer`, `timeOut`.

See also `USBBRICKIO.DEBUG`, `USBBRICKIO.VENDORID`, `USBBRICKIO.PRODUCTID`, `USBBRICKIO.NREADBUFFER`, `USBBRICKIO.NWRITEBUFFER`, `USBBRICKIO.TIMEOUT`

**write** (*brickIO*, *wmsg*)

Writes data to the brick through usb using the hidapi interface.

**Parameters** *wmsg* (*uint8 array*) – Data to be written to the brick via usb

## 1.6 btBrickIO

```
class source.btBrickIO(varargin)
```

**List of methods:**

- `open()`
- `close()`
- `read()`

- `write()`
- `setProperties()`

Bluetooth interface between MATLAB and the brick

## Notes

- Connects to the bluetooth module on the host through a serial connection. Hence be sure that a serial connection to the bluetooth module can be made. Also be sure that the bluetooth module has been paired to the brick before trying to connect.

- **Usage is OS-dependent:**

- Windows: the `deviceName`- & `channel`-properties are needed for connection. The implementation is based on the Instrument Control toolbox.
- Linux (and potentially Mac): `serialPort`-property is needed for connection. The implementation is based on MATLAB's serial port implementation.

- For general information, see also `BrickIO`.

### debug

*bool* – If true, each open/close/read/write-call will be shown in the console. Defaults to false.

### serialPort

*string* – Path to the serial-port object. Only needed when using MATLAB's serial class (i.e. on linux/mac). Defaults to  `'/dev/rfcomm0'`.

### deviceName

*string* – Name of the BT-device = the brick. Only needed when using the Instrument Control toolbox (i.e. on windows). Defaults to  `'EV3'`.

### channel

*numeric > 0* – BT-channel of the connected BT-device. Only needed when using the Instrument Control toolbox (i.e. on windows). Defaults to 1.

### timeOut

*numeric >= 0* – seconds after which a timeout-error occurs if no packet could be read. Defaults to 10.

### backend

`'serial' | 'instrumentControl'`  – Backend this implementation is based on. Is automatically chosen depending on the OS. Defaults to  `'serial'`  on linux/mac systems, and to  `'instrumentControl'`  on windows systems.

## Examples

```
% Connecting on windows commHandle = btBrickIO('deviceName', 'MyEV3', 'channel', 1); % % Connecting on windows using MATLABs default serial port implementation for testing commHandle = btBrickIO('deviceName', 'MyEV3', 'channel', 1, 'backend', 'serial'); % % Connecting on mac/linux commHandle = btBrickIO('serPort', '/dev/rfcomm0');
```

### close (brickIO)

Closes the bluetooth connection the brick using `fclose`.

### open (brickIO)

Opens the bluetooth connection to the brick using `fopen`.

### read (brickIO)

Reads data from the brick through bluetooth via `fread` and returns the data in `uint8` format.

**setProperties** (*brickIO*, *varargin*)

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

The syntax is as follows: `commHandle.setProperties('propertyName1', propertyValue1, 'propertyName2', propertyValue2, ...)`. Valid, optional properties are: `debug`, `serPort`, `deviceName`, `channel`, `timeout`.

See also `BTBRICKIO.DEBUG`, `BTBRICKIO.SERIALPORT`, `BTBRICKIO.DEVICENAME`, `BTBRICKIO.CHANNEL`, `BTBRICKIO.TIMEOUT`

**write** (*brickIO*, *wmsg*)

Writes data to the brick through bluetooth via `fwrite`.

**Parameters** `wmsg` (*uint8 array*) – Data to be written to the brick via bluetooth



**S**

source, [17](#)



## B

backend (source.btBrickIO attribute), 19  
 batteryMode (source.EV3 attribute), 4  
 batteryValue (source.EV3 attribute), 4  
 beep() (source.EV3 method), 4  
 brakeMode (source.Motor attribute), 7  
 btBrickIO (class in source), 18

## C

channel (source.btBrickIO attribute), 19  
 close() (source.btBrickIO method), 19  
 close() (source.hidapi method), 14  
 close() (source.usbBrickIO method), 18  
 connect() (source.EV3 method), 4  
 currentSpeed (source.Motor attribute), 7

## D

debug (source.btBrickIO attribute), 19  
 debug (source.EV3 attribute), 4  
 debug (source.Motor attribute), 7  
 debug (source.Sensor attribute), 12  
 debug (source.usbBrickIO attribute), 18  
 deviceName (source.btBrickIO attribute), 19  
 disconnect() (source.EV3 method), 5

## E

enumerate() (source.hidapi method), 15  
 error() (source.hidapi method), 15  
 EV3 (class in source), 3  
 exit() (source.hidapi method), 15

## G

getHIDInfoString() (source.hidapi method), 15  
 getManufacturersString() (source.hidapi method), 16  
 getProductString() (source.hidapi method), 16  
 getSerialNumberString() (source.hidapi method), 16

## H

handle (source.hidapi attribute), 14  
 hidapi (class in source), 13

## I

init() (source.hidapi method), 16  
 internalReset() (source.Motor method), 8  
 isConnected (source.EV3 attribute), 4  
 isRunning (source.Motor attribute), 7

## L

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

## M

mode (source.Sensor attribute), 10  
 Motor (class in source), 6  
 motorA (source.EV3 attribute), 3  
 motorB (source.EV3 attribute), 3  
 motorC (source.EV3 attribute), 3  
 motorD (source.EV3 attribute), 4

## N

nReadBuffer (source.hidapi attribute), 14  
 nReadBuffer (source.usbBrickIO attribute), 18  
 nWriteBuffer (source.hidapi attribute), 14  
 nWriteBuffer (source.usbBrickIO attribute), 18

## O

open() (source.btBrickIO method), 19  
 open() (source.hidapi method), 16  
 open() (source.usbBrickIO method), 18

## P

playTone() (source.EV3 method), 5  
 power (source.Motor attribute), 7  
 productID (source.hidapi attribute), 14  
 productID (source.usbBrickIO attribute), 18

## R

read() (source.btBrickIO method), 19  
 read() (source.hidapi method), 16  
 read() (source.usbBrickIO method), 18  
 read\_timeout() (source.hidapi method), 16  
 reset() (source.Sensor method), 13  
 resetTachoCount() (source.Motor method), 8

## S

Sensor (class in source), [10](#)  
sensor1 (source.EV3 attribute), [4](#)  
sensor2 (source.EV3 attribute), [4](#)  
sensor3 (source.EV3 attribute), [4](#)  
sensor4 (source.EV3 attribute), [4](#)  
serialPort (source.btBrickIO attribute), [19](#)  
setBrake() (source.Motor method), [8](#)  
setNonBlocking() (source.hidapi method), [17](#)  
setProperties() (source.btBrickIO method), [19](#)  
setProperties() (source.EV3 method), [5](#)  
setProperties() (source.Motor method), [8](#)  
setProperties() (source.Sensor method), [13](#)  
setProperties() (source.usbBrickIO method), [18](#)  
sheader (source.hidapi attribute), [14](#)  
slib (source.hidapi attribute), [14](#)  
smoothStart (source.Motor attribute), [7](#)  
smoothStop (source.Motor attribute), [7](#)  
source (module), [3](#), [6](#), [10](#), [13](#), [17](#), [18](#)  
speedRegulation (source.Motor attribute), [7](#)  
start() (source.Motor method), [9](#)  
stop() (source.Motor method), [9](#)  
stopTone() (source.EV3 method), [6](#)  
syncedStart() (source.Motor method), [9](#)  
syncedStop() (source.Motor method), [9](#)

## T

tachoCount (source.Motor attribute), [7](#)  
timeOut (source.btBrickIO attribute), [19](#)  
timeOut (source.usbBrickIO attribute), [18](#)  
tonePlayed() (source.EV3 method), [6](#)  
type (source.Motor attribute), [7](#)  
type (source.Sensor attribute), [12](#)

## U

usbBrickIO (class in source), [17](#)

## V

value (source.Sensor attribute), [12](#)  
vendorID (source.hidapi attribute), [14](#)  
vendorID (source.usbBrickIO attribute), [18](#)

## W

waitFor() (source.Motor method), [10](#)  
write() (source.btBrickIO method), [20](#)  
write() (source.hidapi method), [17](#)  
write() (source.usbBrickIO method), [18](#)