RWTH - Mindstorms NXT Toolbox

v4.04

List of functions

01.10.2010

http://www.mindstorms.rwth-aachen.de

Functions - Alphabetical List

COM_CloseNXT	Closes and deletes a specific NXT handle, or clears all existing handles
COM_CollectPacket	Reads data from a USB or serial/Bluetooth port, retrieves exactly one packet
COM_CreatePacket	Generates a valid Bluetooth packet ready for transmission (i.e. sets length)
COM_GetDefaultNXT	Returns the global default NXT handle if it was previously set
COM_MakeBTConfigFile	Creates a Bluetooth configuration file (needed for Bluetooth connections)
COM_OpenNXT	Opens USB or Bluetooth connection to NXT device and returns a handle
COM_OpenNXTEx	Opens USB or Bluetooth connection to NXT; advanced version, more options
COM_ReadI2C	Requests and reads sensor data via I2C from a correctly configured digital sensor.
COM_SendPacket	Sends a communication protocol packet (byte-array) via a USB or Bluetooth
COM_SetDefaultNXT	Sets global default NXT handle (will be used by other functions as default)
CalibrateColor	Enables calibration mode of the HiTechnic color sensor V1
CalibrateCompass	Enables calibration mode of the HiTechnic compass sensor
CalibrateEOPD	Calibrates the HiTechnic EOPD sensor (measures/sets calibration matrix)
CalibrateGyro	Calibrates the HiTechnic Gyro sensor (measures/sets an offset while in rest)
CloseSensor	Closes a sensor port (e.g. turns off active light of the light sensor)
DebugMode	Gets or sets debug state (i.e. if textOut prints messages to the command window)
DirectMotorCommand	Sends a direct command to the specified motor
GetAccelerator	Reads the current value of the HiTechnic acceleration sensor
GetColor	Reads the current value of the HiTechnic Color V1 or V2 sensor
GetCompass	Reads the current value of the HiTechnic compass sensor
GetEOPD	Reads the current value of the HiTechnic EOPD sensor
GetGyro	Reads the current value of the HiTechnic Gyro sensor
GetInfrared	Reads the current value of the Hitechnic infrared sensor (infrared seeker)

GetLight	Reads the current value of the NXT light sensor
GetNXT2Color	Reads the current value of the color sensor from the NXT 2.0 set
GetRFID	Reads the transponder ID detected by the Codatex RFID sensor
GetSound	Reads the current value of the NXT sound sensor
GetSwitch	Reads the current value of the NXT switch / touch sensor
GetUltrasonic	Reads the current value of the NXT ultrasonic sensor
MAP_GetCommModule	Reads the IO map of the communication module
MAP_GetInputModule	Reads the IO map of the input module
MAP_GetOutputModule	Reads the IO map of the output module
MAP_GetSoundModule	Reads the IO map of the sound module
MAP_GetUIModule	Reads the IO map of the user interface module
MAP_SetOutputModule	Writes the IO map to the output module
MOTOR_A	Symbolic constant MOTOR_A (returns 0)
MOTOR_B	Symbolic constant MOTOR_B (returns 1)
MOTOR_C	Symbolic constant MOTOR_C (returns 2)
NXC_GetSensorMotorData	Retrieves selected data from all analog sensors and all motors in a single packet
NXC_MotorControl	Sends advanced motor-command to the NXC-program MotorControl on the NXT brick
NXC_ResetErrorCorrection	Sends reset error correction command to the NXC-program MotorControl on the NXT
NXTMotor	Constructs an NXTMotor object
NXT_GetBatteryLevel	Returns the current battery level in milli volts
NXT_GetCurrentProgramName	Returns the name of the current running program
NXT_GetFirmwareVersion	Returns the protocol and firmware version of the NXT
NXT_GetInputValues	Executes a complete sensor reading (requests and retrieves input values)
NXT_GetOutputState	Requests and retrieves an output motor state reading
NXT_LSGetStatus	Gets the number of available bytes for digital low speed sensors (I2C)
NXT_LSRead	Reads data from a digital low speed sensor port (I2C)
NXT_LSWrite	Writes given data to a digital low speed sensor port (I2C)
NXT_MessageRead	Retrieves a "NXT-to-NXT message" from the specified inbox
NXT_MessageWrite	Writes a "NXT-to-NXT message" to the NXT's incoming BT mailbox queue
NXT_PlaySoundFile	Plays the given sound file on the NXT Brick
NXT_PlayTone	Plays a tone with the given frequency and duration
NXT_ReadIOMap	Reads the IO map of the given module ID
NXT_ResetInputScaledValue	Resets the sensor's ScaledVal back to 0 (depends on current sensor mode)
NXT_ResetMotorPosition	Resets NXT internal counter for specified motor, relative or absolute counter

NXT_SendKeepAlive	Sends a KeepAlive packet. Optional: requests sleep time limit.
NXT_SetBrickName	Sets a new name for the NXT Brick (connected to the specified handle)
NXT_SetInputMode	Sets a sensor mode, configures and initializes a sensor to be read out
NXT_SetOutputState	Sends previously specified settings to current active motor.
NXT_StartProgram	Starts the given program on the NXT Brick
NXT_StopProgram	Stops the currently running program on the NXT Brick
NXT_StopSoundPlayback	Stops the current sound playback
NXT_WriteIOMap	Writes the IO map to the given module ID
OpenAccelerator	Initializes the HiTechnic acceleration sensor, sets correct sensor mode
OpenColor	Initializes the HiTechnic color V1 or V2 sensor, sets correct sensor mode
OpenCompass	Initializes the HiTechnic magnetic compass sensor, sets correct sensor mode
OpenEOPD	Initializes the HiTechnic EOPD sensor, sets correct sensor mode
OpenGyro	Initializes the HiTechnic Gyroscopic sensor, sets correct sensor mode
OpenInfrared	Initializes the HiTechnic infrared seeker sensor, sets correct sensor mode
OpenLight	Initializes the NXT light sensor, sets correct sensor mode
OpenNXT2Color	Initializes the LEGO color sensor from the NXT 2.0 set, sets correct sensor mode
OpenRFID	Initializes the Codatex RFID sensor, sets correct sensor mode
OpenSound	Initializes the NXT sound sensor, sets correct sensor mode
OpenSwitch	Initializes the NXT touch sensor, sets correct sensor mode
OpenUltrasonic	Initializes the NXT ultrasonic sensor, sets correct sensor mode
OptimizeToolboxPerformance	Copies binary versions of typecastc to toolbox for better performance
ReadFromNXT	Reads current state of specified motor(s) from NXT brick
ResetPosition	Resets the position counter of the given motor(s).
SENSOR_1	Symbolic constant SENSOR_1 (returns 0)
SENSOR_2	Symbolic constant SENSOR_2 (returns 1)
SENSOR_3	Symbolic constant SENSOR_3 (returns 2)
SENSOR_4	Symbolic constant SENSOR_4 (returns 3)
SendToNXT	Send motor settings to the NXT brick
Stop	Stops or brakes specified motor(s)
StopMotor	Stops / brakes specified motor. (Synchronisation will be

	lost after this)
SwitchLamp	Switches the LEGO lamp on or off (has to be connected to a motor port)
USGetSnapshotResults	Retrieves up to eight echos (distances) stored inside the US sensor
USMakeSnapshot	Causes the ultrasonic sensor to send one snapshot ("ping") and record the echos
WaitFor	Wait for motor(s) to stop (busy waiting)
checkStatusByte	Interpretes the status byte of a return package, returns error message
readFromIniFile	Reads parameters from a configuration file (usually *.ini)
textOut	Wrapper for fprintf() which can optionally write screen output to a logfile

Functions by Category

NXT Communication

COM_CloseNXT	Closes and deletes a specific NXT handle, or clears all existing handles
COM_CollectPacket	Reads data from a USB or serial/Bluetooth port, retrieves exactly one packet
COM_CreatePacket	Generates a valid Bluetooth packet ready for transmission (i.e. sets length)
COM_GetDefaultNXT	Returns the global default NXT handle if it was previously set
COM_MakeBTConfigFile	Creates a Bluetooth configuration file (needed for Bluetooth connections)
COM_OpenNXT	Opens USB or Bluetooth connection to NXT device and returns a handle
COM_OpenNXTEx	Opens USB or Bluetooth connection to NXT; advanced version, more options
COM_ReadI2C	Requests and reads sensor data via I2C from a correctly configured digital sensor.
COM_SendPacket	Sends a communication protocol packet (byte-array) via a USB or Bluetooth
COM_SetDefaultNXT	Sets global default NXT handle (will be used by other functions as default)

NXT Sensors

CalibrateColor	Enables calibration mode of the HiTechnic color sensor V1
CalibrateCompass	Enables calibration mode of the HiTechnic compass sensor
CalibrateEOPD	Calibrates the HiTechnic EOPD sensor (measures/sets calibration matrix)
CalibrateGyro	Calibrates the HiTechnic Gyro sensor (measures/sets an offset while in rest)
CloseSensor	Closes a sensor port (e.g. turns off active light of the light sensor)
GetAccelerator	Reads the current value of the HiTechnic acceleration sensor
GetColor	Reads the current value of the HiTechnic Color V1 or V2 sensor
GetCompass	Reads the current value of the HiTechnic compass sensor
GetEOPD	Reads the current value of the HiTechnic EOPD sensor

GetGyro	Reads the current value of the HiTechnic Gyro sensor
GetInfrared	Reads the current value of the Hitechnic infrared sensor (infrared seeker)
GetLight	Reads the current value of the NXT light sensor
GetNXT2Color	Reads the current value of the color sensor from the NXT 2.0 set
GetRFID	Reads the transponder ID detected by the Codatex RFID sensor
GetSound	Reads the current value of the NXT sound sensor
GetSwitch	Reads the current value of the NXT switch / touch sensor
GetUltrasonic	Reads the current value of the NXT ultrasonic sensor
OpenAccelerator	Initializes the HiTechnic acceleration sensor, sets correct sensor mode
OpenColor	Initializes the HiTechnic color V1 or V2 sensor, sets correct sensor mode
OpenCompass	Initializes the HiTechnic magnetic compass sensor, sets correct sensor mode
OpenEOPD	Initializes the HiTechnic EOPD sensor, sets correct sensor mode
OpenGyro	Initializes the HiTechnic Gyroscopic sensor, sets correct sensor mode
OpenInfrared	Initializes the HiTechnic infrared seeker sensor, sets correct sensor mode
OpenLight	Initializes the NXT light sensor, sets correct sensor mode
OpenNXT2Color	Initializes the LEGO color sensor from the NXT 2.0 set, sets correct sensor mode
OpenRFID	Initializes the Codatex RFID sensor, sets correct sensor mode
OpenSound	Initializes the NXT sound sensor, sets correct sensor mode
OpenSwitch	Initializes the NXT touch sensor, sets correct sensor mode
OpenUltrasonic	Initializes the NXT ultrasonic sensor, sets correct sensor mode
SENSOR_1	Symbolic constant SENSOR_1 (returns 0)
SENSOR_2	Symbolic constant SENSOR_2 (returns 1)
SENSOR_3	Symbolic constant SENSOR_3 (returns 2)
SENSOR_4	Symbolic constant SENSOR_4 (returns 3)
USGetSnapshotResults	Retrieves up to eight echos (distances) stored inside the US sensor
USMakeSnapshot	Causes the ultrasonic sensor to send one snapshot ("ping") and record the echos

NXTMotor Class Methods

NXTMotor	Constructs an NXTMotor object
ReadFromNXT	Reads current state of specified motor(s) from NXT brick
ResetPosition	Resets the position counter of the given motor(s).
SendToNXT	Send motor settings to the NXT brick
Stop	Stops or brakes specified motor(s)
WaitFor	Wait for motor(s) to stop (busy waiting)

Classic NXT Motor Functions

DirectMotorCommand	Sends a direct command to the specified motor
MOTOR_A	Symbolic constant MOTOR_A (returns 0)
MOTOR_B	Symbolic constant MOTOR_B (returns 1)
MOTOR_C	Symbolic constant MOTOR_C (returns 2)
NXC_MotorControl	Sends advanced motor-command to the NXC-program MotorControl on the NXT brick
NXC_ResetErrorCorrection	Sends reset error correction command to the NXC-program MotorControl on the NXT
StopMotor	Stops / brakes specified motor. (Synchronisation will be lost after this)
SwitchLamp	Switches the LEGO lamp on or off (has to be connected to a motor port)

NXT Direct Commands

NXT_GetBatteryLevel	Returns the current battery level in milli volts
NXT_GetCurrentProgramName	Returns the name of the current running program
NXT_GetFirmwareVersion	Returns the protocol and firmware version of the NXT
NXT_GetInputValues	Executes a complete sensor reading (requests and retrieves input values)
NXT_GetOutputState	Requests and retrieves an output motor state reading
NXT_LSGetStatus	Gets the number of available bytes for digital low speed sensors (I2C)
NXT_LSRead	Reads data from a digital low speed sensor port (I2C)
NXT_LSWrite	Writes given data to a digital low speed sensor port (I2C)
NXT_MessageRead	Retrieves a "NXT-to-NXT message" from the specified inbox
NXT_MessageWrite	Writes a "NXT-to-NXT message" to the NXT's incoming BT mailbox queue
NXT_PlaySoundFile	Plays the given sound file on the NXT Brick
NXT_PlayTone	Plays a tone with the given frequency and duration
NXT_ReadIOMap	Reads the IO map of the given module ID
NXT_ResetInputScaledValue	Resets the sensor's ScaledVal back to 0 (depends on current sensor mode)

NXT_ResetMotorPosition	Resets NXT internal counter for specified motor, relative or absolute counter
NXT_SendKeepAlive	Sends a KeepAlive packet. Optional: requests sleep time limit.
NXT_SetBrickName	Sets a new name for the NXT Brick (connected to the specified handle)
NXT_SetInputMode	Sets a sensor mode, configures and initializes a sensor to be read out
NXT_SetOutputState	Sends previously specified settings to current active motor.
NXT_StartProgram	Starts the given program on the NXT Brick
NXT_StopProgram	Stops the currently running program on the NXT Brick
NXT_StopSoundPlayback	Stops the current sound playback
NXT_WriteIOMap	Writes the IO map to the given module ID

NXT Module Map Functions

MAP_GetCommModule	Reads the IO map of the communication module
MAP_GetInputModule	Reads the IO map of the input module
MAP_GetOutputModule	Reads the IO map of the output module
MAP_GetSoundModule	Reads the IO map of the sound module
MAP_GetUIModule	Reads the IO map of the user interface module
MAP_SetOutputModule	Writes the IO map to the output module

General Functions

checkStatusByte	Interpretes the status byte of a return package, returns error message
OptimizeToolboxPerformance	Copies binary versions of typecastc to toolbox for better performance
readFromIniFile	Reads parameters from a configuration file (usually *.ini)
textOut	Wrapper for fprintf() which can optionally write screen output to a logfile

Debug Functions

DebugMode	Gets or sets debug state (i.e. if textOut prints messages to the command window)

Future Functions

	Retrieves selected data from all analog sensors and all

NXC_GetSensorMotorData motors in a single packet

COM_CloseNXT

Closes and deletes a specific NXT handle, or clears all existing handles

Contents

- Syntax
- Description
- Limitations
- Example
- See also
- Signature

Syntax

```
COM_CloseNXT(handle)

COM_CloseNXT('all')

COM_CloseNXT('all', inifilename)
```

Description

After using NXT handles, a user should free the device (and the memory occupied by the handle) by calling this method. After the clean up invoked by this call, an NXT brick can be accessed and used again (by `COM_OpenNXT` or `COM_OpenNXTEx`.

`COM_CloseNXT(handle)` will close and erase the specified device. `handle` has to be a valid handle struct created by either `COM_OpenNXT` or `COM_OpenNXTEx`.

`COM_CloseNXT('all')` will close and erase all existing NXT devices from memory (as long as the toolbox could keep track of them). All USB handles will be destroyed, all open serial ports (for Bluetooth connections) will be closed. This can be useful at the beginning of a program to create a "fresh start" and a well-defined starting environment. Please note that a `clear all` command can cause this function to fail (in such a way, that not all open USB devices can be closed, since all information about them has be cleare from MATLAB's memory). If this happens, an NXT device might appear to be busy and cannot be used. Usually rebooting the NXT helps, if not try to restart MATLAB as well. So be careful with using `clear all` before |COM_CloseNXT('all').

`COM_CloseNXT('all', inifilename)` will do the same as above, but instead of closing all open serial ports, only the COM-Port specified in `inifilename` will be used (a valid Bluetooth configuration file can be created by the function `COM_MakeBTConfigFile`). This syntax helps to avoid interference with other serial ports that might be used by other (MATLAB) programs at the same time. Note that still all open USB devices will be closed.

Limitations

If you call `COM_CloseNXT('all')` after a `clear all` command has been issued, the function will not be able to close all remaining open USB handles, since they have been cleared out of memory. This is a problem on Linux systems. You will not be able to use the NXT device without rebooting it. Solution: Either use only `clear` in your programs, or you use the `COM_CloseNXT('all')` statement before `clear all`. The best way however is to track your

handles carefully and close them manually before exiting whenever possible!

Example

```
handle = COM_OpenNXT('bluetooth.ini', check');
COM_SetDefaultNXT(handle);
NXT_PlayTone(440,10);
COM_CloseNXT(handle);
```

See also

COM_OpenNXT, **COM_OpenNXTEx**, **COM_MakeBTConfigFile**, **COM_SetDefaultNXT**,

Signature

- **Author:** Linus Atorf (see AUTHORS)
- **Date:** 2009/08/31
- **Copyright:** 2007-2010, RWTH Aachen University

Published with wg_publish; V1.0

COM_CollectPacket

Reads data from a USB or serial/Bluetooth port, retrieves exactly one packet

Contents

- Syntax
- Description
- Example
- See also
- Signature

Syntax

```
[type cmd statusbyte content] = COM_CollectPacket(handle)
```

```
[type cmd statusbyte content] = COM_CollectPacket(handle, 'dontcheck')
```

Description

`[type cmd statusbyte content] = COM_CollectPacket(handle)` reads one packet on the communication channel specified by the `handle` struct (PC system: handle struct containing e.g. serial handle, Linux system: handle struct containing file handle). The USB / Bluetooth handle struct can be obtained by the `COM_OpenNXT` or `COM_GetDefaultNXT` command. The return value `type` specifies the telegram type according to the LEGO Mindstorms communication protcol. The `cmd` value determines the specific command. `status` indicates if an error occured on the NXT brick. The function `checkStatusByte` is interpreting this information per default. The `content` column vector represents the remaining payload of the whole return packet. E.g. it contains the current battery level in milli volts, that then has to be converted to a valid integer from its byte representation (i.e. using `wordbytes2dec`).

`[type cmd statusbyte content] = COM_CollectPacket(handle, 'dontcheck')` disables the validation check of the `status` value by function `checkStatusBytes`.

varargin : set to 'dontcheck' if the status byte should not automatically be checked. Only use this if you expect error messages. Possible usage is for LSGetStatus, where this can happen...

For more details about the syntax of the return packet see the LEGO Mindstorms communication protocol.

Note:

This function uses the specific Bluetooth settings from the ini-file that was specified when opening the handle. Parameters used here are `SendSendPause` and `SendReceivePause`, which will cause this function to wait a certain amount of milliseconds between each consecutive send or receive operation to avoid packet loss or buffer overflows inside the blutooth stack.

Example

```
COM_MakeBTConfigFile();

handle = COM_OpenNXT('bluetooth.ini');

[type cmd] = name2commandbytes('KEEPALIVE');
content = [];  % no payload in NXT command KEEPALIVE
packet = COM_CreatePacket(type, cmd, 'reply', content);

COM_SendPacket(packet, handle);

[type cmd statusbyte content] = COM_CollectPacket(handle);
% Now you could check the statusbyte or interpret the content.
% Or maybe check for valid type and cmd before...
```

See also

COM_CreatePacket, **COM_SendPacket**, **COM_OpenNXT**, **COM_GetDefaultNXT**, **COM_MakeBTConfigFile**, **checkStatusByte**,

Signature

- **Author:** Linus Atorf (see AUTHORS)
- **Date:** 2009/08/31
- **Copyright:** 2007-2010, RWTH Aachen University

Published with wg_publish; V1.0

COM_CreatePacket

Generates a valid Bluetooth packet ready for transmission (i.e. sets length)

Contents

- Syntax
- Description
- Example
- See also
- Signature

Syntax

```
bytes = COM_CreatePacket(CommandType, Command, ReplyMode, ContentBytes)
```

Description

`bytes = COM_CreatePacket(CommandType, Command, ReplyMode, ContentBytes)` creates a valid Bluetooth packet conform to the LEGO Mindstorms communication protocol. The `CommandType` specifies the telegram type (direct or system command (see the LEGO Mindstorms communication protocol documentation for more details)). This type is determined from the function `name2commandbytes`. The `Command` specifies the actual command according to the LEGO Mindstorms communication protocol. By the `ReplyMode` one can request an acknowledgement for the packet transmission. The two strings `'reply'` and `'dontreply'` are valid. The content byte-array is given by the `ContentBytes`.

The return value `bytes` represents the complete Bluetooth packet conform to the LEGO Mindstorms Communication protocol.

Note:

The activated `ReplyMode` should only be used if it is necessary. According to the official LEGO statement "Testing during development has shown that the Bluetooth Serial Port communication has some disadvantages when it comes to streaming data. ... One problem is a time penalty (of around 30ms) within the Bluecore chip when switching from receive-mode to transmit-mode. ... To handle the problem of the time penalty within the Bluecore chip, users should send data using Bluetooth without requesting a reply package. This will mean that the Bluecore chip won't have to switch direction for every received package and will not incur a 30ms penalty for every data package."

Example

```
[type cmd] = name2commandbytes('PLAYTONE');
content(1:2) = dec2wordbytes(frequency, 2);
content(3:4) = dec2wordbytes(duration, 2);

packet = COM_CreatePacket(type, cmd, 'dontreply', content);
```

See also

COM_SendPacket, COM_CollectPacket, name2commandbytes, dec2wordbytes,

Signature

- **Author:** Linus Atorf (see AUTHORS)
- **Date:** 2008/07/09
- **Copyright:** 2007-2010, RWTH Aachen University

Published with wg_publish; V1.0

COM_GetDefaultNXT

Returns the global default NXT handle if it was previously set

Contents

- Syntax
- Description
- Example
- See also
- Signature

Syntax

```
h = COM_GetDefaultNXT()
```

Description

`h = COM_GetDefaultNXT()` returns the global default NXT handle `h` if it was previously set. The default global NXT handle is used by all NXT-Functions per default if no other handle is specified. To set this global handle the function `COM_SetDefaultNXT` is used.

Example

```
handle = COM_OpenNXT('bluetooth.ini');
COM_SetDefaultNXT(handle);
MyNXT = COM_GetDefaultNXT();
% now MyNXT and handle refer to the same device
```

See also

COM_SetDefaultNXT, COM_OpenNXT, COM_OpenNXTEx,

Signature

- **Author:** Linus Atorf (see AUTHORS)
- **Date:** 2008/07/07
- **Copyright:** 2007-2010, RWTH Aachen University

Published with wg_publish; V1.0

COM_MakeBTConfigFile

Creates a Bluetooth configuration file (needed for Bluetooth connections)

Contents

- Syntax
- Description
- Example
- See also
- Signature

Syntax

```
COM_MakeBTConfigFile()
```

Description

`COM_MakeBTConfigFile()` starts a user guided dialog window to select the output directory, the file name and the Bluetooth paramters like e.g. COM port.

The little program creates a specific Bluetooth configuration file for the current PC system. Make sure the configuration file is accessible under MATLAB if you try to open a Bluetooth connection using `COM_OpenNXT` and the correct file name.

Example

```
COM_MakeBTConfigFile();
handle = COM_OpenNXT('bluetooth.ini');
```

See also

COM_OpenNXT, COM_CloseNXT, COM_OpenNXTEx,

Signature

- **Author:** Alexander Behrens, Linus Atorf (see AUTHORS)
- **Date:** 2008/07/09
- **Copyright:** 2007-2010, RWTH Aachen University

Published with wg_publish; V1.0

COM_OpenNXT

Opens USB or Bluetooth connection to NXT device and returns a handle

Contents

- Syntax
- Description
- Limitations of COM_CloseNXT
- Example
- See also
- Signature

Syntax

```
handle = COM_OpenNXT()
```

```
handle = COM_OpenNXT(inifilename)
```

Description

`handle = COM_OpenNXT()` tries to open a connection via USB. The first NXT device that is found will be used. Device drivers (Fantom on Windows, libusb on Linux) have to be already installed for USB to work.

`handle = COM_OpenNXT(inifilename)` will search the USB bus for NXT devices, just as the syntax without any parameters. If this fails for some reason (no USB connection to the NXT available, no device drivers installed, or NXT device is busy), the function will try to establish a connection via Bluetooth, using the given Bluetooth configuration file (you can create one easily with `COM_MakeBTConfigFile`.

Note that this function is the most simple way to get an NXT handle. If you need a method to access multiple NXTs or more options, see the advanced function `COM_OpenNXTEx`. In fact, `COM_OpenNXT` is just a convenient wrapper to `COM_OpenNXTEx('Any', ...)`.

Limitations of COM_CloseNXT

If you call `COM_CloseNXT('all')` after a `clear all` command has been issued, the function will not be able to close all remaining open USB handles, since they have been cleared out of memory. This is a problem on Linux systems. You will not be able to use the NXT device without rebooting it. Solution: Either use only `clear` in your programs, or you use the `COM_CloseNXT('all')` statement before `clear all`. The best way however is to track your handles carefully and close them manually (`COM_CloseNXT(handle)`) before exiting whenever possible!

Example

```
handle = COM_OpenNXT('bluetooth.ini');
COM_SetDefaultNXT(handle);
NXT_PlayTone(440,10);
COM_CloseNXT(handle);
```

See also

COM_OpenNXTEx, **COM_CloseNXT**, **COM_MakeBTConfigFile**, **COM_SetDefaultNXT**,

Signature

- **Author:** Linus Atorf (see AUTHORS)
- **Date:** 2009/07/10
- **Copyright:** 2007-2010, RWTH Aachen University

Published with wg_publish; V1.0

COM_OpenNXTEx

Opens USB or Bluetooth connection to NXT; advanced version, more options

Contents

- Syntax
- Description
- Limitations of COM_CloseNXT
- Examples
- See also
- Signature

Syntax

```
handle = COM_OpenNXTEx('USB', UseThisNXTMAC)

handle = COM_OpenNXTEx('Bluetooth', UseThisNXTMAC, inifilename)

handle = COM_OpenNXTEx('Any', UseThisNXTMAC, inifilename)

handle = COM_OpenNXTEx('USB' , UseThisNXTMAC, 'MotorControlFilename', motorcontrolfile)

handle = COM_OpenNXTEx('Bluetooth', UseThisNXTMAC, inifilename, 'MotorControlFilename',
motorcontrolfile)

handle = COM_OpenNXTEx('Any' , UseThisNXTMAC, inifilename, 'MotorControlFilename',
motorcontrolfile)
```

Description

This function establishes a connection to an NXT brick and returns the handle structure that has to be used with NXT-functions (you can call `COM_SetDefaultNXT(handle)` afterwards for easier use).

For a more convenient way to open an NXT handle with less parameters, the function `COM_OpenNXT` is provided.

Different types of connection modes are supported. In all modes, you can set `UseThisNXTMAC` to a string with the NXT's MAC address (serial number). A connection will then only be estabslished to a matching NXT brick. This can be useful for programs with multiple NXT devices. Set it to an empty string `''` to use any NXT available (usually the first one found). The string can have the format '001612345678' or '00:16:12:34:56:78'.

```
handle = COM_OpenNXTEx('USB', UseThisNXTMAC)
```

This will try to open a connection via USB. Device drivers (Fantom on Windows, libusb on Linux) have to be installed.

```
handle = COM_OpenNXTEx('Bluetooth', UseThisNXTMAC, inifilename)
```

Uses Bluetooth as communication method. A valid inifile containing parameters like the COM-Port has to be specified in `inifilename`. To create an inifile with Bluetooth settings, the function `COM_MakeBTConfigFile` is available.

Note that as of right now, the parameter `UseThisNXTMAC` will be ignored for Bluetooth connections until implemented in a future version.

```
handle = COM_OpenNXTEx('Any', UseThisNXTMAC, inifilename)
```

This syntax combines the two parameter settings from above. `inifilename` has to be given. The function will try to locate an NXT device on the USB bus first. If this fails for some reason (no USB connection to the NXT available, no device drivers installed, or NXT device is busy), the function will silently try to establish a connection via Bluetooth.

The advantage is that this version works with both Bluetooth and USB connections *without changing* any code. Plug or unplug the USB cable to switch between connection types...

The optional string-parameter `'MotorControlFilename'` can be used to override the default file name for the embedded NXC program MotorControl, which will be launched by the method. Specify `'MotorControlFilename'`, followed by a the actual filename to be started in `motorcontrolfile`. You can set this to any executable file present on the NXT. The filename conventions are the same as for `NXT_StartProgram`. Set it to an empty string `''` to disable the embedded MotorControl program. In this case, the class `NXTMotor` will not completely be available for usage. This option is intended for advanced users.

Limitations of COM_CloseNXT

If you call `COM_CloseNXT('all')` after a `clear all` command has been issued, the function will not be able to close all remaining open USB handles, since they have been cleared out of memory. This is a problem on Linux systems. You will not be able to use the NXT device without rebooting it. Solution: Either use only `clear` in your programs, or you use the `COM_CloseNXT('all')` statement before `clear all`. The best way however is to track your handles carefully and close them manually (`COM_CloseNXT(handle)`) before exiting whenever possible!%

Examples

```
myNXT = COM_OpenNXTEx('Any', '001612345678', 'bluetooth.ini');
% This will connect to an NXT device with the MAC/serial number 001612345678,
% first trying via USB. If this fails (no drivers installed or no matching USB
% device found), a connection via Bluetooth will be established, using
% the paramters found in the given config file.
```

```
myNXT = COM_OpenNXTEx('USB', '', 'MotorControlFilename, 'MyProgram.rxe');
% This will try to connect to an NXT device via USB, using the first
% one found (we set |UseThisNXTMAC| to |''|). Instead of the embedded
% default MotorControl program, a custom user file MyProgram.rxe will
% try to be launched...
```

See also

COM_OpenNXT, **COM_CloseNXT**, **COM_MakeBTConfigFile**, **COM_SetDefaultNXT**,

Signature

- **Author:** Linus Atorf (see AUTHORS)
- **Date:** 2009/08/31
- **Copyright:** 2007-2010, RWTH Aachen University

Published with wg_publish; V1.0

COM_ReadI2C

Requests and reads sensor data via I2C from a correctly configured digital sensor.

Contents

- Syntax
- Description
- Example
- See also
- Signature

Syntax

```
ReturnBytes = COM_ReadI2C(Port, RequestLen, DeviceAddress, RegisterAddress)

ReturnBytes = COM_ReadI2C(Port, RequestLen, DeviceAddress, RegisterAddress, handle)
```

Description

This function is used to retrieve data from digital sensors (like the ultrasonic) in a comfortable way. It is designed as a helping function for developers wanting to access new sensors. For already implemented sensors (e.g. ultrasound, as well as HiTechnic's acceleration and infrared sensors), use the provided high-level functions such as `GetUltrasonic`, `GetInfrared`, etc.

For I2C communication, usually the `NXT_SetInputMode` command has to be used with the `LOWSPEED_9V` or `LOWSPEED` setting. Afterwards, commands can be send with `NXT_LSWrite`. Once a sensor is correctly working, i.e. has data available, you can use this function to retrieve them.

In `COM_ReadI2C(Port, RequestLen, DeviceAddress, RegisterAddress)`, `Port` is the sensor-port the sensor is connected to. `RequestLen` specifies the amount of bytes you want to retrieve. For ultasound, this is 1. `DeviceAddress` is the sensor's address on the I2C bus. This sometimes can be changed, but not for the ultrasonic sensor. Default value is 0x02 (2 in decimal). Finally, `RegisterAddress` is the address where you want to read data from. For the ultrasound and many other sensors, the "data section" starts at 0x42 (66 in decimal).

As last argument you can pass a valid NXT-handle to be used by this function. If no handle is passed, the default set by `COM_SetDefaultNXT` will be used.

Returns: `ReturnBytes`, byte-array (column vector) of uint8. This array contains the raw sensor-data you requested. How to interpret them depends on the sensor. If communication failed (even after automatic retransmission) -- e.g. when the sensor get's disconnected while in use -- an empty vector `[]` will be returned.

Note:

Please note that the return values of this function are of type uint8. You have to convert them to double (using `double()`) before performing calculations with them, otherwise you might get unexpected results!

The sensor you are addressing with this command has to be correctly opened and

initialized of course -- otherwise no valid data can be received.

Example

This example opens and reads the ultrasonic sensor

```
port = SENSOR_1;
handle = COM_OpenNXT('bluetooth.ini');

OpenUltrasonic(port);

% retrieve 1 byte from device 0x02, register 0x42
data = COM_ReadI2C(port, 1, uint8(2), uint8(66));

if isempty(data)
    DistanceCM = -1;
else
    % don'f forget this double()!!!
    DistanceCM = double(data(1));
end%if
```

See also

NXT_LSWrite, **NXT_LSRead**, **NXT_LSGetStatus**, **NXT_SetInputMode**, **OpenUltrasonic**, **GetUltrasonic**, **SENSOR_1**, **SENSOR_2**, **SENSOR_3**, **SENSOR_4**,

Signature

- **Author:** Linus Atorf (see AUTHORS)
- **Date:** 2008/09/23
- **Copyright:** 2007-2010, RWTH Aachen University

Published with wg_publish; V1.0

COM_SendPacket

Sends a communication protocol packet (byte-array) via a USB or Bluetooth

Contents

- Syntax
- Description
- Example
- See also
- Signature

Syntax

`COM_SendPacket(Packet, handle)`

Description

`COM_SendPacket(Packet, handle)` sends the given byte-array `Packet` (column vector), (which can easily be created by the function `COM_CreatePacket`) over the USB or Bluetooth channel specified by the given `handle` (struct) created by the function `COM_OpenNXT` or `COM_OpenNXTEx`, or obtained from `COM_GetDefaultNXT`.

Note:

In the case of a Bluetooth connection this function uses the specific settings from the ini-file that was specified when opening the handle. Parameters used here are `SendSendPause` and `SendReceivePause`, which will cause this function to wait a certain amount of milliseconds between each consecutive send or receive operation to avoid packet loss or buffer overflows inside the blutooth stack.

Example

```
COM_MakeBTConfigFile();

handle = COM_OpenNXT('bluetooth.ini');

[type cmd] = name2commandbytes('KEEPALIVE');
content = [];  % no payload in NXT command KEEPALIVE
packet = COM_CreatePacket(type, cmd, 'dontreply', content);

COM_SendPacket(packet, bt_handle);
```

See also

COM_CreatePacket, COM_CollectPacket, COM_OpenNXT, COM_GetDefaultNXT, COM_MakeBTConfigFile,

Signature

- **Author:** Linus Atorf (see AUTHORS)
- **Date:** 2009/08/31
- **Copyright:** 2007-2010, RWTH Aachen University

COM_SetDefaultNXT

Sets global default NXT handle (will be used by other functions as default)

Contents

- Syntax
- Description
- Example
- See also
- Signature

Syntax

`COM_SetDefaultNXT(h)`

Description

`COM_SetDefaultNXT(h)` sets the given handle `h` to the global NXT handle, which is used by all NXT-Functions per default if no other handle is specified. To create and open an NXT handle (Bluetooth or USB), the functions `COM_OpenNXT` and `COM_OpenNXTEx` can be used. To retrieve the global default handle user `COM_GetDefaultNXT`.

Example

```
MyNXT = COM_OpenNXT('bluetooth.ini');
COM_SetDefaultNXT(MyNXT);
```

See also

COM_GetDefaultNXT, COM_OpenNXT, COM_OpenNXTEx,

Signature

- **Author:** Linus Atorf (see AUTHORS)
- **Date:** 2008/07/07
- **Copyright:** 2007-2010, RWTH Aachen University

Published with wg_publish; V1.0

CalibrateColor

Enables calibration mode of the HiTechnic color sensor V1

Contents

- Syntax
- Description
- Example
- See also
- Signature

Syntax

```
CalibrateColor(port, mode)

CalibrateColor(port, mode, handle)
```

Description

Do not use this function with the HiTechnic Color Sensor V2. It has a bright white flashing LED. This function is intended for the HiTechnic Color Sensor V1 (milky, weak white LED).

Calibrate the color sensor with white and black reference value. It's not known whether calibration of the color sensor makes sense. HiTechnic doku says nothing, some people say it is necessary, but it works and has effect ;-). The sensor LEDs make a short flash after successful calibration. When calibrated, the sensor keeps this information in non-volatile memory. There are two different modes for calibration:

- `mode` = 1: white balance calibration Puts the sensor into white balance calibration mode. For best results the sensor should be pointed at a diffuse white surface at a distance of approximately 15mm before calling this method. After a fraction of a second the sensor lights will flash and the calibration is done.

- `mode` = 2: black level calibration Puts the sensor into black/ambient level calibration mode. For best results the sensor should be pointed in a direction with no obstacles for 50cm or so. This reading the sensor will use as a base level for other readings. After a fraction of a second the sensor lights will flash and the calibration is done. When calibrated, the sensor keeps this information in non-volatile memory.

The color sensor has to be opened (using `OpenColor`) before execution.

The given `port` number specifies the connection port. The value `port` can be addressed by the symbolic constants `SENSOR_1` , `SENSOR_2`, `SENSOR_3` and `SENSOR_4` analog to the labeling on the NXT Brick.

The last optional argument can be a valid NXT handle. If none is specified, the default handle will be used (call `COM_SetDefaultNXT` to set one).

Example

```
% color must be open for calibration
OpenColor(SENSOR_2);
```

```
% white calibration mode
CalibrateColor(SENSOR_2, 1);
% pause for changing position
pause(5);
% black calibration mode
CalibrateColor(SENSOR_2, 2);
```

See also

OpenColor, **GetColor**, **CloseSensor**,

Signature

- **Author:** Rainer Schnitzler, Linus Atorf (see AUTHORS)
- **Date:** 2010/09/16
- **Copyright:** 2007-2010, RWTH Aachen University

Published with wg_publish; V1.0

CalibrateCompass

Enables calibration mode of the HiTechnic compass sensor

Contents

- Syntax
- Description
- Example
- See also
- Signature

Syntax

```
CalibrateCompass(port, f_start)

CalibrateCompass(port, f_start, handle)
```

Description

Calibrate the compass to reduce influence of metallic objects, especially of the NXT motor and brick on compass values. You have to calibrate a roboter only once until the design changes. During calibration the compass should make two full rotations very slowly. The compass sensor has to be opened (using `OpenCompass`) before execution.

Set `f_start = true` to start calibration mode, and `f_start = false` to stop it. In between those commands, the calibration (compass rotation) should occur.

The given `port` number specifies the connection port. The value `port` can be addressed by the symbolic constants `SENSOR_1` , `SENSOR_2`, `SENSOR_3` and `SENSOR_4` analog to the labeling on the NXT Brick.

The last optional argument can be a valid NXT handle. If none is specified, the default handle will be used (call `COM_SetDefaultNXT` to set one).

Example

```
% compass must be open for calibration
OpenCompass(SENSOR_2);

% enable calibration mode
CalibrateCompass(SENSOR_2, true);

% compass is attached to motor A, rotate 2 full turns
m = NXTMotor('A', 'Power', 5, 'TachoLimit', 720)
m.SendToNXT();

m.WaitFor();

% calibration should now be complete!
CalibrateCompass(SENSOR_2, false);
```

See also

OpenCompass, GetCompass, CloseSensor, NXT_LSRead, NXT_LSWrite,

Signature

- **Author:** Rainer Schnitzler, Linus Atorf (see AUTHORS)
- **Date:** 2008/08/01
- **Copyright:** 2007-2010, RWTH Aachen University

Published with wg_publish; V1.0

CalibrateEOPD

Calibrates the HiTechnic EOPD sensor (measures/sets calibration matrix)

Contents

- Syntax
- Description
- Limitations
- Examples
- See also
- Signature

Syntax

```
CalibrateEOPD(port, 'NEAR', nearDist)

CalibrateEOPD(port, 'FAR', farDist)

calibMatrix = CalibrateEOPD(port, 'READMATRIX')

CalibrateEOPD(port, 'SETMATRIX', calibMatrix)
```

Description

To help you make sense of the HiTechnic EOPD sensor values, this function can calibrate the sensor. The method is based on this article: http://www.hitechnic.com/blog/eopd-sensor/eopd-how-to-measure-distance/#more-178

Before your start calibration, open the sensor using `OpenEOPD` and a mode of your choice. Please note: The calibration will be valid for this mode only. So if you choose long range mode during calibration, you must use this mode all the time when working with this specific calibration setting.

The calibration process is straight forward. You place the sensor at a known distance in front of a surface. First you need to chose a short distance, e.g. around 3cm (not too close). Then you call this function with `calibrationMode = 'NEAR'`, followed by `nearDist` set to the actual distance. This can be centimeters, millimeters, or LEGO studs. The unit doesn't matter as long as you keep it consistend. The value later returned by `GetEOPD` will be in this exact units.

As second step, you have to place the sensor at another known distance, preferrably at the end of the range. Let's just say we use 9cm this time. Now call this functions with `calibrationMode = 'FAR'`, followed by a 9. That's it. The sensor is now calibrated.

Before you continue to use the sensor, you should retrieve the calibration matrix and store it for later use. This matrix is essentialy just a combination of the two distances you used for calibration, and the according EOPD raw sensor readings. Out of these two data pairs, the distance mapping is calculated, which is used inside `GetEOPD`. To retrieve the matrix, call `calibMatrix = CalibrateEOPD(port, 'READMATRIX')`.

If later on you want to leave out the calibration of a specific EOPD sensor for certain environmental conditions, you can simply re-use the calibration matrix. Call

`CalibrateEOPD(port, 'SETMATRIX', calibMatrix)`. The format of the 2x2 calibMatrix is: `[nearDist nearEOPD; farDist farEOPD]`.

To summarize:

1. Use the `'NEAR'` mode with a short distance to the surface.
2. Use the `'FAR'` mode with a long distance to the surface (all relatively. The order can be swapped).
3. Retrieve and store the calibration matrix using the `'READMATRIX'` mode.
4. Later on, if you want to skip steps 1 - 3, just directly load the matrix from step 3 using the `'SETMATRIX'` mode.

Limitations

Calibration is stored inside the NXT handle, for a specific port. This means after closing the NXT handle, or when connecting the sensor to another port, calibration is lost. That is why you should either always run the calibration at the begin of your program, or restore the previous state with the `'SETMATRIX'` calibration mode.

Unlike most other functions, this one cannot be called with an NXT handle as last argument. Please use `COM_SetDefaultNXT` before.

Examples

```
port = SENSOR_2;
OpenEOPD(port, 'SHORT');

% place sensor to 3cm distance, you can also try 2cm or similar
CalibrateEOPD(port, 'NEAR', 3);
pause;

% place sensor to 9cm distance, you can also try 10cm or similar
CalibrateEOPD(port, 'FAR', 9);

% retrieve & display calibration matrix
calibMatrix = CalibrateEOPD(port, 'READMATRIX');
disp(calibMatrix);

% now the sensor can be used
[dist raw] = GetEOPD(port);

% clean up, as usual. LED stays on anyway
CloseSensor(port);
```

```
% Later on in another program, you can
% restore the calibration:
OpenEOPD(port, 'SHORT'); % use same mode as for calibration

% manually set calibMatrix or load from file

% now restore calibration
CalibrateEOPD(port, 'SETMATRIX', calibMatrix);

% sensor ready to be used now...
```

See also

OpenEOPD, **GetEOPD**, **CloseSensor**, **NXT_SetInputMode**, **NXT_GetInputValues**,

Signature

- **Author:** Linus Atorf (see AUTHORS)

- **Date:** 20010/09/22
- **Copyright:** 2007-2010, RWTH Aachen University

Published with wg_publish; V1.0

CalibrateGyro

Calibrates the HiTechnic Gyro sensor (measures/sets an offset while in rest)

Contents

- Syntax
- Description
- Limitations
- Examples
- See also
- Signature

Syntax

```
offset = CalibrateGyro(port, 'AUTO')

offset = CalibrateGyro(port, 'AUTO', handle)

offset = CalibrateGyro(port, 'MANUAL', manualOffset)

offset = CalibrateGyro(port, 'MANUAL', manualOffset, handle)
```

Description

In order to use the HiTechnic Gyro Sensor, it has first to be opened using `OpenGyro`. Then `CalibrateGyro` should be called (or a warning will be issued). Only after this you can safely use `GetGyro` to retrieve values.

This function will set (and return) the new offset (i.e. reading during rest) of the according Gyro sensor. Normally users should use the

automatic calibration mode: `offset = CalibrateGyro(port, 'AUTO')`

The offset will be calculated automatically. During this function the Gyro sensor value will be measured for at least 1 second (or for at least 5 times). During this period, the sensor must be at full rest!

If you want to save time during your program with a well-known Gyro sensor, or you cannot assure that the sensor is at rest during calibration, you can use the automatic calibration once in the command line and remember the determined offset value. Using manual calibration, you can then set a hardcoded value manually (saving you time and the calibration for this sensor in the future in that specific program).

Use `CalibrateGyro(port, 'MANUAL', manualOffset)` to achieve this, with a correct offset obtained from automatic calibration. This call won't require that the sensor doesn't move. Also the call is very fast (as compared to at least 1 second in automatic mode). Use integers for `manualOffset`, as the gyro sensor is only accurate to +/- 1 degree per second anyway.

The last optional argument (for both modes) can be a valid NXT handle. If none is specified, the default handle will be used (call `COM_SetDefaultNXT` to set one).

Note:

Manual calibration only works for one specific sensor (i.e. one unique piece of hardware). Other sensors might have different offsets. Also it could be possible that the offset changes over time or is dependent on your working environment (humidity, temperature, etc).

Limitations

Calibration is stored inside the NXT handle, for a specific port. This means after closing the NXT handle, or when connecting the sensor to another port, calibration is lost. That is why you should either always run the calibration at the begin of your program, or restore the previous offset with the `'MANUAL'` calibration mode.

Examples

```
% in this example the gyro is used with automatic
% calibration, very straight forward

port = SENSOR_2;
OpenGyro(port);
CalibrateGyro(port, 'AUTO');

% now the gyro is ready to be used!
% do something, main program etc...
speed = GetGyro(port);

% do something else, loop etc...
% don't forget to clean up
CloseSensor(port);
```

```
% in this example we save the time and effort of
% automatic calibration each time the main program is run...
% on a command window, type:
h = COM_OpenNXT();
COM_SetDefaulNXT(h);
OpenGyro(SENSOR_1);
% now, once the automatic calibration:
offset = CalibrateGyro(SENSOR_1, 'AUTO');
% remember this value...
```

```
% our main program looks like this:
% always open gyro first:
OpenGyro(SENSOR_1);
% now use the offset value determined earlier:
CalibrateGyro(SENSOR_1, 'MANUAL', offset);
% ready to use GetGyro now...
```

See also

OpenGyro, **GetGyro**, **CloseSensor**, **NXT_SetInputMode**, **NXT_GetInputValues**,

Signature

- **Author:** Linus Atorf (see AUTHORS)
- **Date:** 2009/04/14
- **Copyright:** 2007-2010, RWTH Aachen University

Published with wg_publish; V1.0

CloseSensor

Closes a sensor port (e.g. turns off active light of the light sensor)

Contents

- Syntax
- Description
- Examples
- See also
- Signature

Syntax

```
CloseSensor(port)

CloseSensor(port, handle)
```

Description

`CloseSensor(port)` closes the sensor `port` opened by the `Open...` functions. The value `port` can be addressed by the symbolic constants `SENSOR_1` , `SENSOR_2`, `SENSOR_3` and `SENSOR_4` analog to the labeling on the NXT Brick. Closing the light sensor deactives the active light mode (the red light is turned off), closing the Ultrasonic sensor stops sending out ultrasound.

The last optional argument can be a valid NXT handle. If none is specified, the default handle will be used (call `COM_SetDefaultNXT` to set one).

Examples

```
    OpenLight(SENSOR_3, 'ACTIVE');
    light = GetLight(SENSOR_3);
    CloseSensor(SENSOR_3);
```

See also

NXT_SetInputMode, OpenLight, OpenSound, OpenSwitch, OpenUltrasonic, SENSOR_1, SENSOR_2, SENSOR_3, SENSOR_4,

Signature

- **Author:** Linus Atorf, Alexander Behrens (see AUTHORS)
- **Date:** 2007/10/15
- **Copyright:** 2007-2010, RWTH Aachen University

Published with wg_publish; V1.0

DebugMode

Gets or sets debug state (i.e. if textOut prints messages to the command window)

Contents

- Syntax
- Description
- Example
- See also
- Signature

Syntax

```
state = DebugMode();

DebugMode(state);
```

Description

The function `textOut` can be used to display text messages inside the command window. These messages can optionally be logged to a file (see textOut for details) or the output can be disable. To turn off those debug messages, the global variable `DisableScreenOut` had to be modified in earlier toolbox versions. Now the function `DebugMode` provides easier access.

`state = DebugMode();` returns the current debug state, the return value is either `'on'` or `'off'`.

`DebugMode(state);` is used to switch between displaying messages and silent mode. The paramter `state` has to be `'on'` or `'off'`.

Note: If you need a fast alternative to `strcmpi(DebugMode(), 'on')`, please consider the private toolbox function `isdebug`.

Example

```
% enable debug messages
DebugMode on
```

```
% remember old setting
oldState = DebugMode();
DebugMode('on');
   % do something with textOut(), it will be displayed!
% restore previous setting
DebugMode(oldState);
```

See also

textOut, isdebug (private),

Signature

- **Author:** Linus Atorf (see AUTHORS)
- **Date:** 2008/07/04
- **Copyright:** 2007-2010, RWTH Aachen University

Published with wg_publish; V1.0

DirectMotorCommand

Sends a direct command to the specified motor

Contents

- Syntax
- Description
- Limitations:
- Examples
- See also
- Signature

Syntax

```
DirectMotorCommand(port, power, angle, speedRegulation, syncedToMotor, turnRatio, rampMode)
```

Description

`DirectMotorCommand(port, power, angle, speedRegulation, syncedToMotor, turnRatio, rampMode)` sends the given settings like motor `port` (`MOTOR_A`, `MOTOR_B` or `MOTOR_C`), the `power` (`-100...100`), the `angle` limit (also called TachoLimit), `speedRegulation` (`'on'`, `'off'`), `syncedToMotor` (`MOTOR_A`, `MOTOR_B`, `MOTOR_C`), `turnRatio` (`-100...100`) and `rampMode` (`'off'`, `'up'`, `'down'`).

This function is basically a convenient wrapper for `NXT_SetOutputState`. It provides the fastest way possible to send a direct command to the motor(s) via Bluetooth or USB. Complex parameter combinations which are needed for speed regulation or synchronous driving when using `NXT_SetOutputState` are not necessary, this function does make sure the motor "just works". See below for examples.

Note:

When driving synced motors, it's recommended to stop the motors between consecutive direct motor command (using `StopMotor`) and to reset their position counters (using `NXT_ResetMotorPosition`).

This function is intended for the advanced user. Knowledge about the LEGO MINDSTORMS NXT Bluetooth Communication Protocol is not required, but can help to understand what this function does.

Limitations:

Generally spoken, using `DirectMotorCommand` together with the class `NXTMotor` (and its method `SendToNXT`) for the same motor is strongly discouraged. This function can interfer with the on-brick embedded NXC program MotorControl and could cause it to crash. It ignores whatever is happening on the NXT when sending the direct command. The only advantage is the low latency.

When using the parameter `angleLimit`, the motor tries to reach the desired position by turning off the power at the specified position. This will lead to overshooting of the motor (i.e. the position it stops will be too high or too low). Additionally, the LEGO firmware applies an error correction mechanism which can lead to confusing results. Please look

inside the chapter "Troubleshooting" of this toolbox documentation for more details.

Examples

```
% let a driving robot go straight a bit.
% we use motor synchronization for ports B & C:
DirectMotorCommand(MOTOR_B, 60, 0, 'off', MOTOR_C, 0, 'off');
pause(5); % driving 5 seconds
StopMotor(MOTOR_B, 'off');
StopMotor(MOTOR_C, 'off');
```

```
% let motor A rotate for 1000 degrees (with COAST after "braking" and
% the firmware's error correction) and with speed regulation:
DirectMotorCommand(MOTOR_A, 50, 1000, 'on', 'off', 0, 'off');
```

```
% this command
DirectMotorCommand(MOTOR_A, 0, 0, 'off', 'off', 0, 'off');
% does exactly the same as calling
StopMotor(MOTOR_A, 'off');
% or as using
m = NXTMotor('A');
m.Stop('off');
```

See also

NXT_SetOutputState, **NXT_GetOutputState**, **NXTMotor**, **SendToNXT**, **Stop**, **StopMotor**,

Signature

- **Author:** Linus Atorf, Alexander Behrens (see AUTHORS)
- **Date:** 2009/08/25
- **Copyright:** 2007-2010, RWTH Aachen University

Published with wg_publish; V1.0

GetAccelerator

Reads the current value of the HiTechnic acceleration sensor

Contents

- Syntax
- Description
- Example
- See also
- Signature

Syntax

```
acc_vector = GetAccelerator(port)

acc_vector = GetAccelerator(port, handle)
```

Description

`acc_vector = GetAccelerator(port)` returns the current 1x3 accelerator vector `acc_vector` of the HiTechnic acceleration sensor. The column vector contains the readings of the x, y, and z-axis, respectively. A reading of 200 is equal to the acceleration of 1g. Maximum range is -2g to +2g. The given `port` number specifies the connection port. The value `port` can be addressed by the symbolic constants `SENSOR_1` , `SENSOR_2`, `SENSOR_3` and `SENSOR_4` analog to the labeling on the NXT Brick.

The last optional argument can be a valid NXT handle. If none is specified, the default handle will be used (call `COM_SetDefaultNXT` to set one).

Example

```
OpenAccelerator(SENSOR_4);
acc_Vector = GetAccelerator(SENSOR_4);
CloseSensor(SENSOR_4);
```

See also

OpenAccelerator, CloseSensor, COM_ReadI2C,

Signature

- **Author:** Linus Atorf (see AUTHORS)
- **Date:** 2008/09/25
- **Copyright:** 2007-2010, RWTH Aachen University

Published with wg_publish; V1.0

GetColor

Reads the current value of the HiTechnic Color V1 or V2 sensor

Contents

- Syntax
- Description
- Limitations
- Example
- See also
- Signature

Syntax

```
[index r g b] = GetColor(port, mode)
```

```
[index r g b] = GetColor(port, mode, handle)
```

Description

This function returns the color index and the RGB-values of the HiTechnic Color sensor. There are two different hardware versions of the sensor.

- The old Color sensor V1 has a single weak LED which is always on once connected. You can spot little red, green and blue lights behind the milky lens. This sensor can be calibrated using the function `CalibrateColor`. It has sometimes trouble getting accurate results. You can use all values for `mode`. Try which one works best for you.
- The new Color sensor V2 has a single bright white LED which is always flashing once connected. The lens is relatively clear. This sensor gives great accuary for most colors, even at some distance. Only `mode = 0` is supported. Other modes will return wrong values. The sensor works fine as it comes, it SHOULD NOT BE CALIBRATED.

The color index values roughly correspond to the following table (when using modes 0 and 1):

```
%           0 = black
%           1 = violet
%           2 = purple
%           3 = blue
%           4 = green
%           5 = lime
%           6 = yellow
%           7 = orange
%           8 = red
%           9 = crimson
%           10 = magenta
%           11 to 16 = pastels
%           17 = white
```

The RGB-Values will be returned depending on the mode parameter.

- `mode = 0` : RGB = the current detection level for the color components red, green and blue in an range of 0 to 255. Use this setting for color sensor V2
- `mode = 1` : RGB = the current relative detection level for the color components red,

green and blue in an range of 0 to 255. The highest value of red, green and blue is set to 255 and the other components are adjusted proportionally. Only V1.

- `mode = 2` : RGB = the analog signal levels for the three color components red, green and blue with an accurancy of 10 bit (0 to 1023). Only V1.
- `mode = 3` : RGB = the current relative detection level for the color components red, green and blue in an range of 0 to 3. Only V1.

The color index (0..63) for mode 2 and mode 3 will return a 6 bit color index number, which encodes red in bit 5 and 4, green in bit 3 and 2 and blue in bit 1 (?).

The given `port` number specifies the connection port. The value `port` can be addressed by the symbolic constants `SENSOR_1` , `SENSOR_2`, `SENSOR_3` and `SENSOR_4` analog to the labeling on the NXT Brick.

For more complex settings the function `COM_ReadI2C` can be used.

The last optional argument can be a valid NXT handle. If none is specified, the default handle will be used (call `COM_SetDefaultNXT` to set one).

Limitations

It's by design that the white LED of the Color sensors cannot be turned off by calling `CloseSensor`. It's always on when the sensor is connected. The V2 hardware version of the sensor performs significantly better than the V1 version.

Example

```
OpenColor(SENSOR_4);
[index r g b] = GetColor(SENSOR_4, 0);
CloseSensor(SENSOR_4);
```

See also

OpenColor, **CalibrateColor**, **CloseSensor**, **OpenNXT2Color**, **GetNXT2Color**, **COM_ReadI2C**,

Signature

- **Author:** Rainer Schnitzler, Linus Atorf (see AUTHORS)
- **Date:** 2010/09/16
- **Copyright:** 2007-2010, RWTH Aachen University

Published with wg_publish; V1.0

GetCompass

Reads the current value of the HiTechnic compass sensor

Contents

- Syntax
- Description
- Example
- See also
- Signature

Syntax

```
degree = GetCompass(port)
```

```
degree = GetCompass(port, handle)
```

Description

`degree = GetCompass(port)` returns the current heading value of the HiTechnic magnetic compass sensor ranging from 0 to 360 where 0 is north and counterclockwise (90 = west etc.). The given `port` number specifies the connection port. The value `port` can be addressed by the symbolic constants `SENSOR_1` , `SENSOR_2`, `SENSOR_3` and `SENSOR_4` analog to the labeling on the NXT Brick.

For more complex settings the functions `NXT_LSRead` and `NXT_LSWrite` can be used.

The last optional argument can be a valid NXT handle. If none is specified, the default handle will be used (call `COM_SetDefaultNXT` to set one).

Example

```
OpenCompass(SENSOR_4);
degree = GetCompass(SENSOR_4);
CloseSensor(SENSOR_4);
```

See also

OpenCompass, CalibrateCompass, CloseSensor, COM_ReadI2C,

Signature

- **Author:** Rainer Schnitzler, Alexander Behrens (see AUTHORS)
- **Date:** 2008/08/01
- **Copyright:** 2007-2010, RWTH Aachen University

Published with wg_publish; V1.0

GetEOPD

Reads the current value of the HiTechnic EOPD sensor

Contents

- Syntax
- Description
- Example
- See also
- Signature

Syntax

```
[calcedDist rawVal] = GetEOPD(port)
```

```
[calcedDist rawVal] = GetEOPD(port, handle)
```

Description

This function returns both a calculated distance and the measured raw value from the HiTechnic EOPD sensor.

The given port number specifies the connection port. The value port can be addressed by the symbolic constants `SENSOR_1`, `SENSOR_2`, `SENSOR_3` and `SENSOR_4` analog to the labeling on the NXT Brick.

Returned raw values are always between 0 and 1023 and indicate reflected light intensity. In `'SHORT'` range mode, the values are usually very low, i.e. < 100 or < 200. For increased sensitivity (`'LONG'` range mode), they can also be higher. This mostly depends on the target surface material.

The `rawVal` output argument is always valid. `calcedDist` however will only have a meaningful value if the sensor is correctly calibrated using `CalibrateEOPD`. Otherwise values might not make sense or are `NaN`. If `rawVal` is 0, `calcedDist` will be set to `Inf`.

More on how to interpret the EOPD sensor values, and a detailed explanation of the calibration formula can be found here: **http://www.hitechnic.com/blog/eopd-sensor/eopd-how-to-measure-distance/#more-178**

The last optional argument can be a valid NXT handle. If none is specified, the default handle will be used (call `COM_SetDefaultNXT` to set one).

Example

```
port = SENSOR_2;
OpenEOPD(port, 'SHORT');

% set calibration matrix
calibMatrix = [3 91; 9 19];
CalibrateEOPD(port, 'SETMATRIX', calibMatrix);

% now the sensor can be used
[dist raw] = GetEOPD(port);

% clean up, as usual. LED stays on anyway
```

```
CloseSensor(port);
```

See also

OpenEOPD, **CalibrateEOPD**, **CloseSensor**, **NXT_SetInputMode**, **NXT_GetInputValues**,

Signature

- **Author:** Linus Atorf (see AUTHORS)
- **Date:** 2010/09/17
- **Copyright:** 2007-2010, RWTH Aachen University

Published with wg_publish; V1.0

GetGyro

Reads the current value of the HiTechnic Gyro sensor

Contents

- Syntax
- Description
- Example
- See also
- Signature

Syntax

```
angularVelocity = GetGyro(port)
```

```
angularVelocity = GetGyro(port, handle)
```

Description

`angularVelocity = GetGyro(port)` returns the current rotational speed detected by the HiTechnic Gyroscopic sensor. Maximum range is from -360 to 360 degrees per second (according to HiTechnic documentation), however greater values have been observed. Returned values are accurate to +/- 1 degree.

Integration over time gives the rotational position (in degrees). Tests give quite good results. The given port number specifies the connection port. The value port can be addressed by the symbolic constants `SENSOR_1`, `SENSOR_2`, `SENSOR_3` and `SENSOR_4` analog to the labeling on the NXT Brick.

Before using this function, the gyro sensor must be initialized using `OpenGyro` and calibrated using `CalibrateGyro`.

The last optional argument can be a valid NXT handle. If none is specified, the default handle will be used (call `COM_SetDefaultNXT` to set one).

Example

```
OpenGyro(SENSOR_2);
CalibrateGyro(SENSOR_2, 'AUTO');
speed = GetGyro(SENSOR_2);
CloseSensor(SENSOR_2);
```

See also

OpenGyro, CalibrateGyro, CloseSensor, NXT_SetInputMode, NXT_GetInputValues,

Signature

- **Author:** Linus Atorf, Rainer Schnitzler (see AUTHORS)
- **Date:** 2010/09/14
- **Copyright:** 2007-2010, RWTH Aachen University

GetInfrared

Reads the current value of the Hitechnic infrared sensor (infrared seeker)

Contents

- Syntax
- Description
- Example
- See also
- Signature

Syntax

```
[direction rawData] = GetInfrared(port)
```

```
[direction rawData] = GetInfrared(port, handle)
```

Description

`[direction rawData] = GetInfrared(port)` returns the current direction an the raw data of the detected infrared signal. `direction` represents the main direction (1-9) calculated based on the measured raw data given in the `rawData` vector (1x5). Five sensors are provided by the infrared seeker. For more information see **http://www.hitechnic.com**

The given `port` number specifies the connection port. The value `port` can be addressed by the symbolic constants `SENSOR_1` , `SENSOR_2`, `SENSOR_3` and `SENSOR_4` analog to the labeling on the NXT Brick.

The last optional argument can be a valid NXT handle. If none is specified, the default handle will be used (call `COM_SetDefaultNXT` to set one).

Example

```
OpenInfrared(SENSOR_4);
[direction rawData] = GetInfrared(SENSOR_4);
CloseSensor(SENSOR_4);
```

See also

OpenInfrared, **CloseSensor**, **COM_ReadI2C**,

Signature

- **Author:** Linus Atorf (see AUTHORS)
- **Date:** 2008/09/25
- **Copyright:** 2007-2010, RWTH Aachen University

Published with wg_publish; V1.0

GetLight

Reads the current value of the NXT light sensor

Contents

- Syntax
- Description
- Examples
- See also
- Signature

Syntax

```
light = GetLight(port)
```

```
light = GetLight(port, handle)
```

Description

`light = GetLight(port)` returns the current light value `light` of the NXT light sensor. The measurement value `light` represents the normalized (default) light value (0..1023 / 10 Bit). The normalized value mode is set per default by the function `OpenLight`. The given `port` number specifies the connection port. The value `port` can be addressed by the symbolic constants `SENSOR_1` , `SENSOR_2`, `SENSOR_3` and `SENSOR_4` analog to the labeling on the NXT Brick.

The last optional argument can be a valid NXT handle. If none is specified, the default handle will be used (call `COM_SetDefaultNXT` to set one).

For more complex settings the function `NXT_GetInputValues` can be used.

Examples

```
OpenLight(SENSOR_1, 'ACTIVE');
light = GetLight(SENSOR_1);
CloseSensor(SENSOR_1);
```

See also

OpenLight, CloseSensor, NXT_GetInputValues, SENSOR_1, SENSOR_2, SENSOR_3, SENSOR_4,

Signature

- **Author:** Linus Atorf, Alexander Behrens (see AUTHORS)
- **Date:** 2010/09/14
- **Copyright:** 2007-2010, RWTH Aachen University

Published with wg_publish; V1.0

GetNXT2Color

Reads the current value of the color sensor from the NXT 2.0 set

Contents

- Syntax
- Description
- Limitations
- Examples
- See also
- Signature

Syntax

```
out = GetNXT2Color(port)
```

```
out = GetNXT2Color(port, handle)
```

Description

This functions retrieves the current value of the LEGO NXT 2.0 Color sensor specified by the sensor port. The value `port` can be addressed by the symbolic constants `SENSOR_1` , `SENSOR_2`, `SENSOR_3` and `SENSOR_4` analog to the labeling on the NXT Brick. This function is intended for the Color sensor that comes with the NXT 2.0 set. It has the label "RGB" written on the front, 3 LED openings (1 black empty spot, the light sensor and a clear lens with tiny red, green, blue LEDs behind it). It is not to be confused with the HiTechnic Color sensors (V1 and V2), for those please see the functions `OpenColor` and `GetColor`.

The sensor has to be opened with `OpenNXT2Color()` before this function can be used.

Depending on the mode the color sensor was opened in, the return value of this function can have one of the following two formats

- **In full color mode** (sensor was opened with `mode = 'FULL'`), the return value will consist of one of the following strings: `'BLACK'`, `'BLUE'`, `'GREEN'`, `'YELLOW'`, `'RED'`, `'WHITE'`. If an error occured, the return value may be `'UNKNOWN'` (unlikely though).
- **In all other modes**, i.e. `'RED'`, `'GREEN'`, `'BLUE'`, `'NONE'`, the returned value will be an integer between 0 and 1023, indicating the amount of reflected / detected light. This is very similar to the behaviour of `GetLight`.

The last optional argument can be a valid NXT handle. If none is specified, the default handle will be used (call `COM_SetDefaultNXT` to set one).

Limitations

The sensor is influenced by ambient light. It reacts differently on daylight than on artificial light. The modes `'RED'` and `'NONE'` are similar to the Light sensor's modes `'ACTIVE'` and `'INACTIVE'`, but the sensors are not perfectly compatible.

Examples

```
port = SENSOR_1;
OpenNXT2Color(port, 'FULL');
color = GetNXT2Color(port);
if strcmp(color, 'BLUE')
    disp('Blue like the ocean');
else
    disp(['The detected color is ' color]);
end%if
CloseSensor(port);
```

```
port = SENSOR_2;
OpenNXT2Color(port, 'NONE');
colorVal = GetNXT2Color(port);
if colorVal > 700
    disp('It''s quite bright!')
end%if
CloseSensor(port);
```

See also

OpenNXT2Color, **CloseSensor**, **OpenColor**, **GetColor**, **OpenLight**, **GetLight**, **COM_ReadI2C**,

Signature

- **Author:** Nick Watts, Linus Atorf (see AUTHORS)
- **Date:** 2010/09/21
- **Copyright:** 2007-2010, RWTH Aachen University

Published with wg_publish; V1.0

GetRFID

Reads the transponder ID detected by the Codatex RFID sensor

Contents

- Syntax
- Description
- Example
- See also
- Signature

Syntax

```
transpID = GetRFID(port)
```

```
transpID = GetRFID(port, handle)
```

Description

`transpID = GetRFID(port)` returns a 5-byte transponder ID (datatype is uint64). The given port number specifies the connection port. The value port can be addressed by the symbolic constants `SENSOR_1`, `SENSOR_2`, `SENSOR_3` and `SENSOR_4` analog to the labeling on the NXT Brick.

The last optional argument can be a valid NXT handle. If none is specified, the default handle will be used (call `COM_SetDefaultNXT` to set one).

Note:

The RFID-tag should be placed in a distance of about 1 to 3cm from the RFID sensor. If the transponder was successfully detected, the orange LED of the sensor will flash. Very rarely, when operating with multiple ID tags close to each other, an ID might not be read correctly (in this case it's usually easy to spot, as it looks very different from the "usual" tag IDs).

Please also note that this function returns about 3 to 5 readings per second when used with USB. Bia Bluetooth however, a single function call can take as long as 1 second, depending on connection quality.

Example

```
OpenRFID(SENSOR_2);
transID = GetRFID(SENSOR_2);
CloseSensor(SENSOR_2);
```

See also

OpenRFID, CloseSensor,

Signature

- **Author:** Linus Atorf, Rainer Schnitzler (see AUTHORS)
- **Date:** 2008/12/1
- **Copyright:** 2007-2010, RWTH Aachen University

Published with wg_publish; V1.0

GetSound

Reads the current value of the NXT sound sensor

Contents

- Syntax
- Description
- Example
- See also
- Signature

Syntax

```
sound = GetSound(port)
```

```
sound = GetSound(port, handle)
```

Description

`sound = GetSound(port)` returns the current sound value `sound` of the NXT sound sensor. The measurement value `sound` represents the normalized (default) sound value (0..1023 / 10 Bit). The normalized value mode is set per default by the function `OpenSound`. The given `port` number specifies the connection port. The value `port` can be addressed by the symbolic constants `SENSOR_1`, `SENSOR_2`, `SENSOR_3` and `SENSOR_4` analog to the labeling on the NXT Brick.

The last optional argument can be a valid NXT handle. If none is specified, the default handle will be used (call `COM_SetDefaultNXT` to set one).

For more complex settings the function `NXT_GetInputMode` can be used.

Example

```
OpenSound(SENSOR_1, 'DB');
sound = GetSound(SENSOR_1);
CloseSensor(SENSOR_1);
```

See also

OpenSound, CloseSensor, NXT_GetInputValues, SENSOR_1, SENSOR_2, SENSOR_3, SENSOR_4,

Signature

- **Author:** Linus Atorf, Alexander Behrens (see AUTHORS)
- **Date:** 2010/09/14
- **Copyright:** 2007-2010, RWTH Aachen University

Published with wg_publish; V1.0

GetSwitch

Reads the current value of the NXT switch / touch sensor

Contents

- Syntax
- Description
- Example
- See also
- Signature

Syntax

```
switch = GetSwitch(port)

switch = GetSwitch(port, handle)
```

Description

`switch = GetSwitch(port)` returns the current switch value `switch` of the NXT switch / touch sensor. The measurement value `switch` represents the pressing mode of the switch / touch sensor. `true` is returned if the switch / touch sensor is being pressed and `false` if it is being released. The given `port` number specifies the connection port. The value `port` can be addressed by the symbolic constants `SENSOR_1`, `SENSOR_2`, `SENSOR_3` and `SENSOR_4` analog to the labeling on the NXT Brick.

The last optional argument can be a valid NXT handle. If none is specified, the default handle will be used (call `COM_SetDefaultNXT` to set one).

For more complex settings the function `NXT_GetInputValues` can be used.

Example

```
OpenSwitch(SENSOR_4);
switchState = GetSwitch(SENSOR_4);
CloseSensor(SENSOR_4);
```

See also

NXT_GetInputValues, OpenSwitch, CloseSensor, SENSOR_1, SENSOR_2, SENSOR_3, SENSOR_4,

Signature

- **Author:** Linus Atorf, Alexander Behrens (see AUTHORS)
- **Date:** 2010/09/14
- **Copyright:** 2007-2010, RWTH Aachen University

Published with wg_publish; V1.0

GetUltrasonic

Reads the current value of the NXT ultrasonic sensor

Contents

- Syntax
- Description
- Limitations
- Example
- See also
- Signature

Syntax

```
distance = GetUltrasonic(port)
```

```
distance = GetUltrasonic(port, handle)
```

Description

`distance = GetUltraSonic(port)` returns the current measurement value `distance` of the NXT ultrasonic sensor. `distance` represents the measured distance in cm. If no echo can be detected (which could indicate that either there is no obstacle in the way, or the ultrasound does not get reflected, e.g. by fur-like surfaces), the reading will be 255. If no measurement can be made (defect sensor, cable disconnected, etc.), a value of -1 will be returned.

The given `port` number specifies the connection port. The value `port` can be addressed by the symbolic constants `SENSOR_1` , `SENSOR_2`, `SENSOR_3` and `SENSOR_4` analog to the labeling on the NXT Brick.

The last optional argument can be a valid NXT handle. If none is specified, the default handle will be used (call `COM_SetDefaultNXT` to set one).

Note:

This function only works when the sensor was correctly opened with `OpenUltrasonic(port)`. If the sensor is being used in snapshot mode, `GetUltrasonic` will not work correctly!

For different uses, see also `OpenUltrasonic(port, 'snapshot')` and the functions `USMakeSnapshot` and `USGetSnapshotResults`.

Limitations

Since the Ultrasonic sensors all operate at the same frequency, multiple US sensors will interfere with each other! If multiple US sensors can "see each other" (or their echos and reflections), results will be unpredictable (and probably also unusable). You can avoid this problem by turning off US sensors, or operating them in snapshot mode (see also `USMakeSnapshot` and `USGetSnapshotResults`).

Due to the speed of sound in air, the ultrasonic sensor needs a certain amount of time to complete a successful measurement. This is why the maximum polling rate has been

limited to 50 Hz (i.e. a call will take 20ms if called too often). This is only relevant for fast USB connections.

Example

```
OpenUltrasonic(SENSOR_4);
distance = GetUltrasonic(SENSOR_4);
CloseSensor(SENSOR_4);
```

See also

OpenUltrasonic, **USMakeSnapshot**, **USGetSnapshotResults**, **CloseSensor**, **NXT_LSRead**, **NXT_LSWrite**,

Signature

- **Author:** Linus Atorf, Alexander Behrens (see AUTHORS)
- **Date:** 2008/01/15
- **Copyright:** 2007-2010, RWTH Aachen University

Published with wg_publish; V1.0

MAP_GetCommModule

Reads the IO map of the communication module

Contents

- Syntax
- Description
- Examples
- See also
- Signature

Syntax

```
map = MAP_GetCommModule()
```

Description

`map = MAP_GetCommModule()` returns the IO map of the communication module. The return value `map` is a struct variable. It contains all communication module information.

Output:

`map.PFunc` % ?

`map.PFuncTwo` % ?

`map.BTPort` % 1x4 cell array contains Bluetooth device information of each NXT Bluetooth port (i = 0..3)

`map.BTPort{i}.BtDeviceTableName` % name of the Bluetooth device

`map.BTPort{i}.BtDeviceTableClassOfDevice` % class of the Bluetooth device

`map.BTPort{i}.BtDeviceTableBdAddr` % MAC address of the Bluetooth device

`map.BTPort{i}.BtDeviceTableDeviceStatus` % status of the Bluetooth device

`map.BTPort{i}.BtConnectTableName` % name of the connected Bluetooth device

`map.BTPort{i}.BtConnectTableClassOfDevice` % class of the connected Bluetooth device

`map.BTPort{i}.BtConnectTablePinCode` % pin code of the connected Bluetooth device

`map.BTPort{i}.BtConnetTableBdAddr` % MAC address of the connected Bluetooth device

`map.BTPort{i}.BtConnectTableHandleNr` % handle nr of the connected Bluetooth device

`map.BTPort{i}.BtConnectTableStreamStatus` % stream status of the connected Bluetooth device

`map.BTPort{i}.BtConnectTableLinkQuality` % link quality of the connected Bluetooth device

`map.BrickDataName` % name of the NXT brick

`map.BrickDataBluecoreVersion` % Bluecore version number

`map.BrickDataBdAddr` % MAC address of the NXT brick

`map.BrickDataBtStateStatus` % Bluetooth state status

`map.BrickDataBtHwStatus` % NXT hardware status

`map.BrickDataTimeOutValue` % time out value

`map.BtDeviceCnt` % number of devices defined within the Bluetooth device table

`map.BtDeviceNameCnt` % number of devices defined within the Bluetooth device table (usually equal to BtDeviceCnt)

`map.HsFlags` % High Speed flags

`map.HsSpeed` % High Speed speed

`map.HsState` % High Speed state

`map.HsSpeed` % High Speed speed

`map.UsbState` % USB state

Examples

```
map = MAP_GetCommModule();
```

See also

NXT_ReadIOMap,

Signature

- **Author:** Alexander Behrens (see AUTHORS)
- **Date:** 2008/05/23
- **Copyright:** 2007-2010, RWTH Aachen University

Published with wg_publish; V1.0

MAP_GetInputModule

Reads the IO map of the input module

Contents

- Syntax
- Description
- Examples
- See also
- Signature

Syntax

```
map = MAP_GetInputModule(port)
```

Description

`map = MAP_GetInputModule(port)` returns the IO map of the input module at the given sensor `port`. The sensor `port` can be addressed by `SENSOR_1`, `SENSOR_2`, `SENSOR_3`, `SENSOR_4` and `'all'`. The return value `map` is a struct variable or cell array (`'all'` mode). It contains all input module information.

Output:

`map.CustomZeroOffset` % custom sensor zero offset value of a sensor.

`map.ADRaw` % raw 10-bit value last read from the ananlog to digital converter. Raw values produced by sensors typically cover some subset of the full 10-bit range.

`map.SensorRaw` % raw sensor value

`map.SensorValue` % tacho/angle limit, 0 means none set

`map.SensorType` % sensor value

`map.SensorMode` % sensor mode

`map.SensorBoolean` % boolean sensor value

`map.DigiPinsDir` % digital pins direction value of a sensor

`map.DigiPinsIn` % digital pins status value of a sensor ?

`map.DigiPinsOut` % digital pins output level value of a sensor

`map.CustomPctFullScale` % custom sensor percent full scale value of the sensor.

`map.CustomActiveStatus` % custom sensor active status value of the sensor

`map.InvalidData` % value of the InvalidData flag of the sensor

Examples

```
map = MAP_GetInputModule(SENSOR_2);
```

```
map = MAP_GetInputModule('all');
```

See also

NXT_ReadIOMap,

Signature

- **Author:** Alexander Behrens (see AUTHORS)
- **Date:** 2008/05/23
- **Copyright:** 2007-2010, RWTH Aachen University

Published with wg_publish; V1.0

MAP_GetOutputModule

Reads the IO map of the output module

Contents

- Syntax
- Description
- Examples
- See also
- Signature

Syntax

```
map = MAP_GetOutputModule(motor)
```

Description

`map = MAP_GetOutputModule(motor)` returns the IO map of the output module at the given `motor` port. The `motor` port can be addressed by `MOTOR_A`, `MOTOR_B`, `MOTOR_C` and `'all'`. The return value `map` is a struct variable or cell array (`'all'` mode). It contains all output module information.

Output:

`map.TachoCount` % internal, non-resettable rotation-counter (in degrees)

`map.BlockTachoCount` % block tacho counter, current motor position, resettable using, ResetMotorAngle (NXT-G counter since block start)

`map.RotationCount` % rotation tacho counter, current motor position (NXT-G counter since program start)

`map.TachoLimit` % tacho/angle limit, 0 means none set

`map.MotorRPM` % current pulse width modulation ?

`map.Flags` % should be always ''. Flags are considered in MAP_SetOutputModule.

`map.Mode` % output mode bitfield 1: MOTORON, 2: BRAKE, 4: REGULATED

`map.ModeName` % output mode name interpreted by output mode bitfield

`map.Speed` % motor power/speed

`map.ActualSpeed` % current actual percentage of full power (regulation mode)

`map.RegPParameter` % proportional term of the internal PID control algorithm

`map.RegIParameter` % integral term of the internal PID control algorithm

`map.RegDParameter` % derivate term of the internal PID control algorithm

`map.RunStateByte` % run state byte

`map.RunStateName` % run state name interpreted by run state byte

`map.RegModeByte` % regulation mode byte

`map.RegModeName` % regulation mode name interpreted by regulation mode byte

`map.Overloaded` % overloaded flag (true: speed regulation is unable to onvercome physical load on the motor)

`map.SyncTurnParam` % current turn ratio, 1: 25%, 2:50%, 3:75%, 4:100% of full volume

Examples

```
map = MAP_GetOutputModule(MOTOR_A);
```

```
map = MAP_GetOutputModule('all');
```

See also

NXT_ReadIOMap,

Signature

- **Author:** Alexander Behrens (see AUTHORS)
- **Date:** 2008/05/22
- **Copyright:** 2007-2010, RWTH Aachen University

Published with wg_publish; V1.0

MAP_GetSoundModule

Reads the IO map of the sound module

Contents

- Syntax
- Description
- Examples
- See also
- Signature

Syntax

```
map = MAP_GetSoundModule()
```

Description

`map = MAP_GetSoundModule()` returns the IO map of the sound module. The return value `map` is a struct variable. It contains all sound module information.

Output:

`map.Frequency` % frequency of the last played ton in Hz

`map.Duration` % duration of the last played ton in ms

`map.SamplingRate` % current sound sample rate

`map.SoundFileName` % sound file name of the last played sound file

`map.Flags` % sound module flag, 'IDLE': sound module is idle, 'UPDATE': a request for plackback is pending, 'RUNNING': playback in progress.

`map.State` % sound module state, 'IDLE'; sound module is idel, 'PLAYING_FILE': sound module is playing a .rso file, 'PLAYING_TONE': a tone is playing, 'STOP': a request to stop playback is in progress.

`map.Mode` % sound module mode, 'ONCE': only play file once , 'LOOP': play file in a loop, 'TONE': play tone.

`map.Volume` % volume: 0: diabled, 1: 25%, 2:50%, 3:75%, 4:100% of full volume

Examples

```
    map = MAP_GetSoundModule();
```

See also

NXT_ReadIOMap,

Signature

- **Author:** Alexander Behrens (see AUTHORS)
- **Date:** 2008/05/22
- **Copyright:** 2007-2010, RWTH Aachen University

Published with wg_publish; V1.0

MAP_GetUIModule

Reads the IO map of the user interface module

Contents

- Syntax
- Description
- Examples
- See also
- Signature

Syntax

```
map = MAP_GetUIModule()
```

Description

`map = MAP_GetUIModule()` returns the IO map of the user interface module. The return value `map` is a struct variable. It contains all user interface module information.

Output:

`map.PMenu` % ?

`map.BatteryVoltage` % battery voltage in mili volt.

`map.LMSfilename` % ?

`map.Flags` % flag bitfield

`map.State` % state value

`map.Button` % button value

`map.RunState` % VM run state

`map.BatteryState` % battery level (0..4)

`map.BluetoothState` % bluetooth state bitfield

`map.UsbState` % USB state

`map.SleepTimout` % sleep timeout value in minutes

`map.SleepTimer` % current sleep timer in minutes

`map.Rechargeable` % true if reachargeable battery is used

`map.Volume` % volume level (0..4)

`map.Error` % error value

`map.OBPPointer` % on brick program pointer

`map.ForceOff` % turn off brick if value is true

Examples

```
map = MAP_GetUIModule();
```

See also

NXT_ReadIOMap,

Signature

- **Author:** Alexander Behrens (see AUTHORS)
- **Date:** 2008/05/23
- **Copyright:** 2007-2010, RWTH Aachen University

Published with wg_publish; V1.0

MAP_SetOutputModule

Writes the IO map to the output module

Contents

- Syntax
- Description
- Examples
- See also
- Signature

Syntax

```
MAP_SetOutputModule(motor, map)
```

```
MAP_SetOutputModule(motor, map, varargin)
```

Description

`map = MAP_SetOutputModule(motor, map)` writes the IO `map` to the output module at the given motor `motor`. The `motor` port can be addressed by `MOTOR_A`, `MOTOR_B`, `MOTOR_C`. The `map` structure has to provide all output module information, listed below.

Input:

`map.TachoCount` % internal, non-resettable rotation-counter (in degrees)

`map.BlockTachoCount` % block tacho counter, current motor position, resettable using, ResetMotorAngle (NXT-G counter since block start)

`map.RotationCount` % rotation tacho counter, current motor position (NXT-G counter since program start)

`map.TachoLimit` % current set tacho/angle limit, 0 means none set

`map.MotorRPM` % current pulse width modulation ?

`map.Flags` % update flag bitfield, commits any changing (see also varargin)

`map.Mode` % output mode bitfield 1: MOTORON, 2: BRAKE, 4: REGULATED

`map.Speed` % current motor power/speed

`map.ActualSpeed` % current actual percentage of full power (regulation mode)

`map.RegPParameter` % proportional term of the internal PID control algorithm

`map.RegIParameter` % integral term of the internal PID control algorithm

`map.RegDParameter` % derivate term of the internal PID control algorithm

`map.RunStateByte` % run state byte

`map.RegModeByte`

% regulation mode byte

`map.Overloaded` % overloaded flag (true: speed regulation is unable to onvercome physical load on the motor)

`map.SyncTurnParam` % current turn ratio, 1: 25%, 2:50%, 3:75%, 4:100% of full volume

map = MAP_SetOutputModule(motor, map, varargin) sets the update flags explicit by the given arguments. 'UpdateMode': commits changes to the mode property 'UpdateSpeed': commits changes to the speed property 'UpdateTachoLimit': commits changes to the tacho limit property 'ResetCounter': resets internal movement counters, cancels current goal, and resets internal error-correction system 'UpdatePID': commits changes to PID regulation parameters 'ResetBlockTachoCount': resets block tacho count (block-relative position counter (NXT-G)) 'ResetRotationCount': resets rotation count (program-relative position counter (NXT-G))

Examples

```
MAP_SetOutputModule(MOTOR_A, map);
```

```
map = MAP_GetOutputModule(MOTOR_A);
map.RegPParameter = 20;
MAP_SetOutputModule(MOTOR_A, map, 'UpdatePID');
```

See also

MAP_GetOutputModule, **NXT_WriteIOMap**,

Signature

- **Author:** Alexander Behrens (see AUTHORS)
- **Date:** 2008/05/22
- **Copyright:** 2007-2010, RWTH Aachen University

Published with wg_publish; V1.0

MOTOR_A

Symbolic constant MOTOR_A (returns 0)

Contents

- Syntax
- Description
- Example
- See also
- Signature

Syntax

```
portA = MOTOR_A()
```

Description

`portA = MOTOR_A()` returns `0` as the value `portA`.

Example

```
portA = MOTOR_A()
%result: >> portA = 0
```

See also

MOTOR_B, MOTOR_C,

Signature

- **Author:** Linus Atorf, Alexander Behrens (see AUTHORS)
- **Date:** 2007/10/15
- **Copyright:** 2007-2010, RWTH Aachen University

Published with wg_publish; V1.0

MOTOR_B

Symbolic constant MOTOR_B (returns 1)

Contents

- Syntax
- Description
- Example
- See also
- Signature

Syntax

```
portB = MOTOR_B()
```

Description

`portB = MOTOR_B()` returns `1` as the value `portB`.

Example

```
portB = MOTOR_B()
%result: >> portB = 1
```

See also

MOTOR_A, MOTOR_C,

Signature

- **Author:** Linus Atorf, Alexander Behrens (see AUTHORS)
- **Date:** 2007/10/15
- **Copyright:** 2007-2010, RWTH Aachen University

Published with wg_publish; V1.0

MOTOR_C

Symbolic constant MOTOR_C (returns 2)

Contents

- Syntax
- Description
- Example
- See also
- Signature

Syntax

```
portC = MOTOR_C()
```

Description

`portC = MOTOR_C()` returns `2` as the value `portC`.

Example

```
portC = MOTOR_C()
%result: >> portC = 2
```

See also

MOTOR_A, MOTOR_B,

Signature

- **Author:** Linus Atorf, Alexander Behrens (see AUTHORS)
- **Date:** 2007/10/15
- **Copyright:** 2007-2010, RWTH Aachen University

Published with wg_publish; V1.0

NXC_GetSensorMotorData

Retrieves selected data from all analog sensors and all motors in a single packet

Contents

- Syntax
- Description
- Limitations
- Signature

Syntax

```
[sensorData motorData] = NXC_GetSensorMotorData(handle)
```

Description

This function uses the embedded NXC program MotorControl to retrieve certain data from all analog sensors and all motors connected to the NXT. The sensors must already be opened. The function does no interprete any data for you. The big advantage of this function is that it retrieves all the data within one single Bluetooth / USB packet, which is faster than retrieving all information one by one after each other...

Limitations

This function is not yet implemented and does not return any data!

Signature

- **Author:** Linus Atorf (see AUTHORS)
- **Date:** 2009/07/15
- **Copyright:** 2007-2010, RWTH Aachen University

Published with wg_publish; V1.0

NXC_MotorControl

Sends advanced motor-command to the NXC-program MotorControl on the NXT brick,

Contents

- Syntax
- Description
- Limitations
- See also
- Signature

Syntax

```
NXC_MotorControl(Port, Power, TachoLimit, SpeedRegulation, ActionAtTachoLimit, SmoothStart)

NXC_MotorControl(Port, Power, TachoLimit, SpeedRegulation, ActionAtTachoLimit, SmoothStart,
handle)
```

Description

The NXC-program "MotorControl" must be running on the brick, otherwise this function will not work. It is used to send advanced motor commands to the NXT that can perform better and more precise motor regulation than possible with only classic direct commands.

While one command is being executed (i.e. when the motor is still being controlled if a `TachoLimit` other than 0 was set), this motor cannot accept new commands. Use the NXTMotor classes command `.WaitFor` to make sure the motor has finished it's current operation, before sending a new one. If the NXC-program receives a new command while it is still busy, a warning signal (high beep, then low beep) will be played.

The command `StopMotor` (or NXTMotor's method `.Stop`) is always available to stop a controlled motor-operation, even before the `TachoLimit` is reached.

Input:

- `Port` has to be a port number between 0 and 2, or an array with max. 2 different motors specified.
- `Power` is the power level applied to the motor, value between -100 and 100 (sign changes direction)
- `TachoLimit` - integer from 0 to 999999, specifies the angle in degrees the motor will try to reach, set 0 to run forever. Note that direction is specified by the sign of `Power`.
- `SpeedRegulation` must be `false` for "normal", unregulated motor control. If set to `true`, single motors will be operated in speed regulation mode. This means that the motor will increase its internal power setting to reach a constant turning speed. Use this option when working with motors under varying load. If you'd like to have motor movement with preferrably constant torque, it's advisable to disable this option. In conjunction with multiple motors (i.e. when `Port` is an array of 2 ports), you have to disable `SpeedRegulation`! Multiple motorss will enable synchronization between the two motors. They will run at the same speed as if they were connected through and axle,

leading to straight movement for driving bots.

- `ActionAtTachoLimit` is a string parameter with valid options `'Coast'`, `'Brake'` or `'HoldBrake'`. It specifies how the motor(s) should react when their position counter reaches the set `TachoLimit`. In COAST mode, the motor(s) will simply be turned of when the `TachoLimit` is reached, leading to free movement until slowly stopping (called coasting). The `TachoLimit` won't be met, the motor(s) move way too far (overshooting), depending on their angular momentum. Use BRAKE mode (default) to let the motor(s) automatically slow down nice and smoothly shortly before the `TachoLimit`. This leads to a very high precision, usually the `TachoLimit` is met within +/- 1 degree (depending on the motor load and speed of course). After this braking, power to the motor(s) is turned off when they are at rest. HOLDBRAKE is similar to BRAKE, but in this case, the active brake of the motors stays enabled (careful, this consumes a lot of battery power), causing the motor(s) to actively keep holding their position.

- `SmoothStart` can be set to `true` to smoothly accelerate movement. This "manual ramp up" of power will occur fairly quickly. It's comfortable for driving robots so that they don't loose traction when starting to move. If used in conjunction with `SpeedRegulation` for single motors, after accleration is finished and the full power is applied, the speed regulation can possibly even accelerate a bit more.

- `handle` (optional) defines the given NXT connection. If no handle is specified, the default one (`COM_GetDefaultNXT()`) is used.

Limitations

If you send a command to the NXT without waiting for the previous motor operation to have finished, the command will be dropped (the NXT indicates this with a high and low beep tone). Use NXTMotor classes `WaitFor` to make sure the motor is ready for new commands, or stop the motor using NXTMotor's method `.Stop`.

The option `SmoothStart` in conjunction with `ActionAtTachoLimit == 'Coast'` is not available. As a workaround, disable `SmoothStart` for this mode.

With `ActionAtTachoLimit = 'Coast'` and synchronous driving (two motors), the motors will stay synced after movement (even after `.WaitFor()` has finished). This is by design. To disable the synchonization, just use `StopMotor(port, 'off')`.

See also

WaitForMotor, NXT_SetOutputState, NXT_GetOutputState, NXC_ResetErrorCorrection, MOTOR_A, MOTOR_B, MOTOR_C%

Signature

- **Author:** Linus Atorf (see AUTHORS)
- **Date:** 2009/07/20
- **Copyright:** 2007-2010, RWTH Aachen University

Published with wg_publish; V1.0

NXC_ResetErrorCorrection

Sends reset error correction command to the NXC-program MotorControl on the NXT

Contents

- Syntax
- Description
- Limitations
- Example
- See also
- Signature

Syntax

```
NXC_ResetErrorCorrection(port)
```

```
NXC_ResetErrorCorrection(port, handle)
```

Description

The NXC-program "MotorControl" must be running on the brick, otherwise this function will not work. It is used to sent advanced motor commands to the NXT that can perform better and more precise motor regulation than possible with only classic direct commands.

This function resets the "internal error correction memory" for the according motor. Usually, you cannot move the NXT motors by hand in between two commands that control the motors, since the modified motor position does not match the internal counters any more. This leads to unexpected motor behaviour (when the NXT firmware tries to correct the manual movements you just made). *The problem described here does not occur when working with the* `NXTMotor` *class.*

To work around this problem, it is possible to reset the error correction counter by hand using this function. It will clear the counter `TachoCount`, since this counter is internally attached to the error correction. The counter `BlockTachoCount` (see direct commands specification of the LEGO Mindstorms Bluetooth Developer Kit) will also be reset (since it is used to coordinate multiple motors during synchronous driving).

It is recommended to call this function before using classic direct commands (i.e. like `NXT_SetOutputState`), to get the intuitively expected results.

Input:

`port` has to be a port number between 0 and 2. It can also be an array of valid port-numbers, i.e. `[0; 1]`, `[0; 2]`, `[1; 2]` or `[0; 1; 2]`. The named constants `MOTOR_A` to `MOTOR_C` can be used for clarity (i.e. `port = [MOTOR_A; MOTOR_B]`.

`handle` is optional and determines the NXT handle to be used, if specified. Otherwise the default handle will be used (set using `COM_SetDefaultNXT`).

Limitations

This function is intended for advanced users. It makes no sense to use it together with the

`NXTMotor` class. It can however be useful in between certain `DirectMotorCommand` calls.

Example

```
% This will reset the TachoCount counter for port A on the NXT, which
% also resets the error correction
NXC_ResetErrorCorrection(MOTOR_A);
% Now MOTOR_A behaves as if the NXT was freshly booted up...
% The "personal" position counter (field Position when calling
% a motor object's method ReadFromNXT()) won't be affected through this
% -- it will stay untouched until reset by a motor object's method
% ResetPosition())
```

See also

NXTMotor, **NXC_MotorControl**, **NXT_SetOutputState**, **NXT_GetOutputState**, **NXT_ResetMotorPosition**, **ReadFromNXT**,

Signature

- **Author:** Linus Atorf (see AUTHORS)
- **Date:** 2008/11/12
- **Copyright:** 2007-2010, RWTH Aachen University

Published with wg_publish; V1.0

NXTMotor

Constructs an NXTMotor object

Contents

- Syntax
- Description
- Limitations
- Example:
- See also
- Signature

Syntax

```
M = NXTMotor()

M = NXTMotor(PORT)

M = NXTMotor(PORT, 'PropName1', PropValue1, 'PropName2', PropValue2, ...)
```

Description

`M = NXTMotor(PORT)` constructs an NXTMotor object with motor port `PORT` and default attributes. `PORT` may be either the port number (0, 1, 2 or `MOTOR_A`, `MOTOR_B`, `MOTOR_C`) or a string specifying the port ('A', 'B', 'C'). To have two motors synchronized `PORT` may be a vector of two ports in ascending order.

`M = NXTMotor(PORT, 'PropName1', PropValue1, 'PropName2', PropValue2, ...)` constructs an NXTMotor object with motor port(s) `PORT` in which the given Property name/value pairs are set on the object. All properties can also be set after creation by dot-notation (see example).

Available properties are:

- `Port` - the motor port(s) being used, either a string composed of the letters `'A'`, `'B'`, `'C'`, or a single value or array of the numbers 0, 1, 2. A maximum of 2 motors is allowed. If 2 motors are specified, the bot will drive in sync mode, good for driving straight ahead.
- `Power` - integer from -100 to 100, sets power level and direction of rotation (0 to 100%)
- `SpeedRegulation` - if set to `true` (default), the motor will try to hold a constant speed by adjusting power output according to load (e.g. friction) - this is only valid for single motors. It must be deactivated when using two motors! If you'd like to have motor movement with preferrably constant torque, it's advisable to disable this option. `SpeedRegulation` must be `false` for "normal", unregulated motor control. If set to `true`, single motors will be operated in speed regulation mode. This means that the motor will increase its internal power setting to reach a constant turning speed. Use this option when working with motors under varying load. If you'd like to have motor movement with preferrably constant torque, it's advisable to disable this option. In conjunction with multiple motors (i.e. when `Port` is an array of 2 ports), you have to

disable SpeedRegulation! Multiple motors will enable synchronization between the two motors. They will run at the same speed as if they were connected through and axle, leading to straight movement for driving bots.

- TachoLimit - integer from 0 to 999999, specifies the angle in degrees the motor will try to reach, set 0 to run forever. Note that direction is specified by the sign of Power.
- ActionAtTachoLimit is a string parameter with valid options 'Coast', 'Brake' or 'HoldBrake'. It specifies how the motor(s) should react when their position counter reaches the set TachoLimit.
  - In COAST mode, the motor(s) will simply be turned of when the TachoLimit is reached, leading to free movement until slowly stopping (called coasting). The TachoLimit won't be met, the motor(s) move way too far (overshooting), depending on their angular momentum.
  - Use BRAKE mode (default) to let the motor(s) automatically slow down nice and smoothly shortly before the TachoLimit. This leads to a very high precision, usually the TachoLimit is met within +/- 1 degree (depending on the motor load and speed of course). After this braking, power to the motor(s) is turned off when they are at rest.
  - HOLDBRAKE is similar to BRAKE, but in this case, the active brake of the motors stays enabled (careful, this consumes a lot of battery power), causing the motor(s) to actively keep holding their position.
- SmoothStart can be set to true to smoothly accelerate movement. This "manual ramp up" of power will occur fairly quickly. It's comfortable for driving robots so that they don't loose traction when starting to move. If used in conjunction with SpeedRegulation for single motors, after accleration is finished and the full power is applied, the speed regulation can possibly even accelerate a bit more. This option is only available for TachoLimit > 0 and ActionAtTachoLimit = 'Brake' or 'HoldBrake'.

For a list of valid methods, see the "See also" section below.

Note:

When using a motor object with two ports set, the motors will be operated in synchronous mode. This means an internal regulation of the NXT firmware tries to move both motors at the same speed and to the same position (so that driving robots can go a straight line for example). With ActionAtTachoLimit == 'Coast' the sync mode will stay enabled during coasting, allowing for the firmware to correct the robot's position (align it straight ahead again). If you want to use those motors again, you must reset/stop the synchonization before by sending a .Stop() to the motors!

Limitations

If you send a command to the NXT without waiting for the previous motor operation to have finished, the command will be dropped (the NXT indicates this with a high and low beep tone). Use the classes method WaitFor to make sure the motor is ready for new commands, or stop the motor using the method .Stop.

The option SmoothStart in conjunction with ActionAtTachoLimit == 'Coast' is not available. As a workaround, disable SmoothStart for this mode. SmoothStart will generally only work when TachoLimit > 0 is set.

With ActionAtTachoLimit = 'Coast' and synchronous driving (two motors), the motors will stay synced after movement (even after `.WaitFor()` has finished). This is by design. To disable the synchonization, just use `.Stop('off')`.

`SpeedRegulation = true` does not always produce the expected result. Due to internal PID regulation, the actually achieved speed can vary or oscillate when using very small values for `Power`. This happens especially when using the motor with a heavy load for small speeds. In this case it can be better to disable `SpeedRegulation`. In general, speed regulation should only be enabled if a constant rotational velocity is desired. For constant torque, better disable this feature.

Example:

```
% Construct a NXTMotor object on port 'B' with a power of
% 60, disabled speed regulation, a TachoLimit of 360 and
% send the motor settings to the NXT brick.
motorB = NXTMotor('B', 'Power', 60)

motorB.SpeedRegulation     = false;
motorB.TachoLimit          = 360;
motorB.ActionAtTachoLimit  = 'Brake'; % this is the default anyway
motorB.SmoothStart         = true;

% enough setting up params, let's go!
motorB.SendToNXT();
% let MATLAB wait until the motor has stopped moving
motorB.WaitFor();

% Play tone when motor is ready to be used again
NXT_PlayTone(400,500);
```

```
% let's use a driving robot
m = NXTMotor('BC', 'Power', 60);
m.TachoLimit          = 1000;
m.SmoothStart         = true,     % start soft
m.ActionAtTachoLimit = 'coast'; % we want very smooth "braking", too :-)
m.SendToNXT();                    % go!

m.WaitFor();                      % are we there yet?

% we're here, motors are still moving / coasting, so give the bot time!
pause(3);

% you can still hear the synchronization (high noisy beeping)
% before we go back, we have to disable the synchronization quickly
m.Stop();

% reverse direction
m.Power = -m.Power;
m.SendToNXT();
m.WaitFor();
pause(3);
m.Stop();

NXT_PlayTone(500, 100); % all done
```

See also

SendToNXT, **ReadFromNXT**, **WaitFor**, **Stop**, **ResetPosition**, **DirectMotorCommand**,

Signature

- **Author:** Linus Atorf, Aulis Telle, Alexander Behrens (see AUTHORS)
- **Date:** 2009/08/24
- **Copyright:** 2007-2010, RWTH Aachen University

NXT_GetBatteryLevel

Returns the current battery level in milli volts

Contents

- Syntax
- Description
- Examples
- See also
- Signature

Syntax

```
voltage = NXT_GetBatteryLevel()

voltage = NXT_GetBatteryLevel(handle)
```

Description

`voltage = NXT_GetBatteryLevel()` returns the current battery level `voltage` of the NXT Brick in milli voltage.

`voltage = NXT_GetBatteryLevel(handle)` uses the given Bluetooth connection `handle`. This should be a serial handle on a PC system and a file handle on a Linux system.

If no Bluetooth handle is specified the default one (`COM_GetDefaultNXT`) is used.

Examples

```
    voltage = NXT_GetBatteryLevel();
```

```
    handle = COM_OpenNXT('bluetooth.ini');
    voltage = NXT_GetBatteryLevel(handle);
```

See also

COM_GetDefaultNXT, NXT_SendKeepAlive,

Signature

- **Author:** Linus Atorf (see AUTHORS)
- **Date:** 2007/10/15
- **Copyright:** 2007-2010, RWTH Aachen University

Published with wg_publish; V1.0

NXT_GetCurrentProgramName

Returns the name of the current running program

Contents

- Syntax
- Description
- Examples
- See also
- Signature

Syntax

```
[name prog_run] = NXT_GetCurrentProgramName()

[name prog_run] = NXT_GetCurrentProgramName(handle)
```

Description

`[name prog_run] = NXT_GetCurrentProgramName()` returns the `name` of the current running program. and the boolean flag `prog_run` (false == no program is running).

`[name prog_run] = NXT_GetCurrentProgramName(handle)` uses the given NXT `handle`.

If no NXT handle is specified the default one (`COM_GetDefaultNXT`) is used.

Examples

```
name = NXT_GetCurrentProgramName();
```

```
handle = COM_OpenNXT('bluetooth.ini');
name = NXT_GetCurrentProgramName(handle);
```

See also

NXT_StartProgram, NXT_StopProgram,

Signature

- **Author:** Alexander Behrens (see AUTHORS)
- **Date:** 2008/10/18
- **Copyright:** 2007-2010, RWTH Aachen University

Published with wg_publish; V1.0

NXT_GetFirmwareVersion

Returns the protocol and firmware version of the NXT

Contents

- Syntax
- Description
- Examples
- See also
- Signature

Syntax

```
[protocol_version firmware_version] = NXT_GetFirmwareVersion()

[protocol_version firmware_version] = NXT_GetFirmwareVersion(handle)
```

Description

`[protocol_version firmware_version] = NXT_GetFirmwareVersion()` returns the protocol and firmware version of the NXT as strings.

`[protocol_version firmware_version] = NXT_GetFirmwareVersion(handle)` uses the given NXT connection `handle`. This should be a struct containing a serial handle on a PC system and a file handle on a Linux system.

If no NXT `handle` is specified the default one (`COM_GetDefaultNXT`) is used.

Examples

```
[protocol_version firmware_version] = NXT_GetFirmwareVersion();
```

```
handle = COM_OpenNXT('bluetooth.ini');
[protocol_version firmware_version] = NXT_GetFirmwareVersion(handle);
```

See also

COM_GetDefaultNXT,

Signature

- **Author:** Alexander Behrens (see AUTHORS)
- **Date:** 2008/05/22
- **Copyright:** 2007-2010, RWTH Aachen University

Published with wg_publish; V1.0

NXT_GetInputValues

Executes a complete sensor reading (requests and retrieves input values)

Contents

- Syntax
- Description
- Examples
- See also
- Signature

Syntax

```
data = NXT_GetInputValues(port)
```

```
data = NXT_GetInputValues(port, handle)
```

Description

`data = NXT_GetInputValues(port)` processes a complete sensor reading, i.e. requests input values and collects the answer of the given sensor `port`. The value `port` can be addressed by the symbolic constants `SENSOR_1`, `SENSOR_2`, `SENSOR_3` and `SENSOR_4` analog to the labeling on the NXT Brick. The return value `data` is a struct variable. It contains several sensor settings and information.

`data = NXT_GetInputValues(port, handle)` uses the given Bluetooth connection `handle`. This should be a serial handle on a PC system and a file handle on a Linux system.

If no Bluetooth handle is specified the default one (`COM_GetDefaultNXT`) is used.

Output:

`data.Port` % current port number (`0..3`)

`data.Valid` % validation flag

`data.Calibrated` % boolean, `true` if calibration file found and used

`data.TypeByte` % sensor type

`data.TypeName` % sensor mode

`data.ModeByte` % mode

`data.ModeName` % mode name

`data.RawADVal` % raw A/D value

`data.NormalizedADVal` % normalized A/D value

`data.ScaledVal` % scaled value

`data.CalibratedVal` % calibrated value

Note:

Data are only valid if `.Valid` is ~= 0. This should usually be the case, but a short while after setting a new sensor mode using `NXT_SetInputMode`, you have to carefully check .Valid on your own! Experience shows that only `.ScaledVal` is influenced by this, apparently `.NormalizedADVal` seems valid all the time, but closer examination is needed...

For more details see the official LEGO Mindstorms communication protocol.

Examples

```
data = NXT_GetInputValues(SENSOR_3);
```

```
handle = COM_OpenNXT('bluetooth.ini');
data = NXT_GetInputValues(SENSOR_1, handle);
```

See also

NXT_SetInputMode, **GetLight**, **GetSwitch**, **GetSound**, **GetUltrasonic**, **SENSOR_1**, **SENSOR_2**, **SENSOR_3**, **SENSOR_4**,

Signature

- **Author:** Linus Atorf (see AUTHORS)
- **Date:** 2007/10/15
- **Copyright:** 2007-2010, RWTH Aachen University

Published with wg_publish; V1.0

NXT_GetOutputState

Requests and retrieves an output motor state reading

Contents

- Syntax
- Description
- Examples
- See also
- Signature

Syntax

```
data = NXT_GetOutputState(port)
```

```
data = NXT_GetOutputState(port, handle)
```

Description

`data = NXT_GetOutputState(port)` requests and retrieves an output motor state reading of the given motor `port`. The value `port` can be addressed by the symbolic constants `MOTOR_A`, `MOTOR_B` and `MOTOR_C` analog to the labeling on the NXT Brick. The return value `data` is a struct variable. It contains several motor settings and information.

`data = NXT_GetOutputState(port, handle)` uses the given Bluetooth connection `handle`. This should be a serial handle on a PC system and a file handle on a Linux system.

If no Bluetooth handle is specified the default one (`COM_GetDefaultNXT`) is used.

Output:

`data.Port` % current port number (`0..3`)

`data.Power` % current motor power

`data.Mode` % motor mode byte

`data.ModeIsMOTORON` % flag: "motor is on"

`data.ModeIsBRAKE` % flag: "motor uses the advanced brake mode (PVM)"

`data.ModeIsREGULATED` % flag: "motor uses a regulation"

`data.RegModeByte` % motor regulation byte

`data.RegModeName` % name of regulation mode

`data.TurnRatio` % turn ratio value

`data.RunStateByte` % motor run state byte

`data.RunStateName` % name of run state

`data.TachoLimit`

% tacho / angle limit

`data.TachoCount` % current absolute tacho count

`data.BlockTachoCount` % current relative tacho count

`data.RotationCount` % current second relative tacho count

For more details see the official LEGO Mindstorms communication protocol.

Examples

```
data = NXT_GetOutputState(MOTOR_B);
```

```
handle = COM_OpenNXT('bluetooth.ini');
data = NXT_GetOutputState(MOTOR_A, handle);
```

See also

ReadFromNXT, **NXT_SetOutputState**, **DirectMotorCommand**, **MOTOR_A**, **MOTOR_B**, **MOTOR_C**,

Signature

- **Author:** Linus Atorf (see AUTHORS)
- **Date:** 2007/10/15
- **Copyright:** 2007-2010, RWTH Aachen University

Published with wg_publish; V1.0

NXT_LSGetStatus

Gets the number of available bytes for digital low speed sensors (I2C)

Contents

- Syntax
- Description
- Examples
- See also
- Signature

Syntax

```
[BytesReady status] = NXT_LSGetStatus(port)
```

```
[BytesReady status] = NXT_LSGetStatus(port, handle)
```

Description

`[BytesReady status] = NXT_LSGetStatus(port)` gets the number of available bytes from the low speed (digital) sensor reading of the given sensor `port`. The value `port` can be addressed by the symbolic constants `SENSOR_1`, `SENSOR_2`, `SENSOR_3` and `SENSOR_4` analog to the labeling on the NXT Brick. The return value `BytesReady` contains the number of bytes available to read. `status` indicates if an error occures by the packet transmission. Function `checkStatusBytes` is interpreting this information per default.

`[BytesReady status] = NXT_LSGetStatus(port, handle)` uses the given Bluetooth connection `handle`. This should be a serial handle on a PC system and a file handle on a Linux system.

If no Bluetooth handle is specified the default one (`COM_GetDefaultNXT`) is used.

For more details see the official LEGO Mindstorms communication protocol.

Note:

This function's status byte sometimes contains an error message: "Pending communication transaction in progress". This is by design (see documentation of NXTCommLSCheckStatus on page 70 of the "LEGO Mindstorms NXT Executable File Specification" document).

Before using LS commands, the sensor mode has to be set to `LOWSPEED_9V` using the NXT_SetInputMode command.

Examples

```
[BytesReady status] = NXT_LSGetStatus(SENSOR_3);
```

```
handle = COM_OpenNXT('bluetooth.ini');
NXT_SetInputMode(SENSOR_1, 'LOWSPEED_9V', 'RAWMODE', 'dontreply');
% note that status can contain errorsmessages, use checkStatusByte
[BytesReady status] = NXT_LSGetStatus(SENSOR_1, handle);
```

See also

NXT_SetInputMode, checkStatusByte, NXT_LSWrite, NXT_LSRead,

Signature

- **Author:** Linus Atorf (see AUTHORS)
- **Date:** 2007/10/15
- **Copyright:** 2007-2010, RWTH Aachen University

Published with wg_publish; V1.0

NXT_LSRead

Reads data from a digital low speed sensor port (I2C)

Contents

- Syntax
- Description
- Examples
- See also
- Signature

Syntax

```
[data BytesRead] = NXT_LSRead(port)

[data BytesRead] = NXT_LSRead(port, handle)

[data BytesRead optionalStatusByte] = NXT_LSRead(port)

[data BytesRead optionalStatusByte] = NXT_LSRead(port, handle)
```

Description

`[data BytesRead] = NXT_LSRead(port))` gets the `data` of the low speed (digital) sensor value of the given sensor `port`. The value `port` can be addressed by the symbolic constants `SENSOR_1`, `SENSOR_2`, `SENSOR_3` and `SENSOR_4` analog to the labeling on the NXT Brick. The return value `BytesRead` contains the number of bytes available to read.

`[data BytesRead] = NXT_LSRead(port, handle)` uses the given Bluetooth connection `handle`. This should be a serial handle on a PC system and a file handle on a Linux system.

`[data BytesRead optionalStatusByte] = NXT_LSRead(port, [handle])` will ignore the automatic statusbyte check and instead return it as output argument. This causes the function to ignore erronous I2C calls or crashes if the sensor is not yet ready. You can effectively save a call to `NXT_LSGetStatus` with this, if you interpret the statusbytes correctly. This may vary, depending on your I2C sensor. The `handle` argument is still optional, like above.

If no NXT handle is specified the default one (`COM_GetDefaultNXT`) is used.

For more details see the official LEGO Mindstorms communication protocol.

Note:

For LS communication on the NXT, data lengths are limited to 16 bytes per command. Furthermore, this protocol does not support variable-length return packages, so the response will always contain 16 data bytes, with invalid data bytes padded with zeros.

Before using LS commands, the sensor mode has to be set to `LOWSPEED_9V` using the NXT_SetInputMode command.

Examples

```
    handle = COM_OpenNXT('bluetooth.ini');
```

```
NXT_SetInputMode(SENSOR_1, 'LOWSPEED_9V', 'RAWMODE', 'dontreply');
% usually we would use NXT_LSWrite before, to request some sort of reply
[data BytesRead] = NXT_LSRead(SENSOR_1, handle);
```

See also

NXT_SetInputMode, **NXT_LSWrite**, **NXT_LSGetStatus**, **COM_ReadI2C**,

Signature

- **Author:** Linus Atorf (see AUTHORS)
- **Date:** 2007/10/15
- **Copyright:** 2007-2010, RWTH Aachen University

Published with wg_publish; V1.0

NXT_LSWrite

Writes given data to a digital low speed sensor port (I2C)

Contents

- Syntax
- Description
- Example
- See also
- Signature

Syntax

```
NXT_LSWrite(port, RXLength, data, ReplyMode)
```

```
NXT_LSWrite(port, RXLength, data, ReplyMode, handle)
```

Description

`NXT_LSWrite(port, RXLength, data, ReplyMode)` writes the given `data` to a low speed (digital) sensor of the given sensor `port`. The value `port` can be addressed by the symbolic constants `SENSOR_1`, `SENSOR_2`, `SENSOR_3` and `SENSOR_4` analog to the labeling on the NXT Brick. The value `RXLength` represents the data length of the expected receiving packet. By the `ReplyMode` one can request an acknowledgement for the packet transmission. The two strings `'reply'` and `'dontreply'` are valid.

`NXT_LSWrite(port, RXLength, data, ReplyMode, handle)` uses the given Bluetooth connection `handle`. This should be a serial handle on a PC system and a file handle on a Linux system.

If no Bluetooth handle is specified the default one (`COM_GetDefaultNXT`) is used.

For more details see the official LEGO Mindstorms communication protocol.

Note:

For LS communication on the NXT, data lengths are limited to 16 bytes per command. Rx Data Length MUST be specified in the write command since reading from the device is done on a master-slave basis.

Before using LS commands, the sensor mode has to be set to `LOWSPEED_9V` using the NXT_SetInputMode command.

Example

```
RequestLen = 1;
I2Cdata = hex2dec(['02'; '42']); % specific ultrasonic I²C command

handle = COM_OpenNXT('bluetooth.ini');
NXT_SetInputMode(SENSOR_1, 'LOWSPEED_9V', 'RAWMODE', 'dontreply');

NXT_LSWrite(SENSOR_1, RequestLen, I2Cdata, 'dontreply', handle);
```

See also

NXT_SetInputMode, **NXT_LSRead**, **NXT_LSGetStatus**, **COM_ReadI2C**,

Signature

- **Author:** Linus Atorf (see AUTHORS)
- **Date:** 2007/10/15
- **Copyright:** 2007-2010, RWTH Aachen University

NXT_MessageRead

Retrieves a "NXT-to-NXT message" from the specified inbox

Contents

- Syntax
- Description
- Examples
- See also
- Signature

Syntax

```
[message localInboxReturn] = NXT_MessageRead(LocalInbox, RemoteInbox, RemoveFromRemote)

[message localInboxReturn] = NXT_MessageRead(LocalInbox, RemoteInbox, RemoveFromRemote, handle)

[message localInboxReturn statusByte] = NXT_MessageRead(LocalInbox, RemoteInbox,
RemoveFromRemote)

[message localInboxReturn statusByte] = NXT_MessageRead(LocalInbox, RemoteInbox,
RemoveFromRemote, handle)
```

Description

This function reads a NXT-to-NXT bluetooth message from a mailbox queue on the NXT. `LocalInbox` and `RemoteInbox` are the mailbox numbers and must be between 0 and 9. The difference between local and remote mailbox is not fully understood, so it's best to use the same value for both parameters. For more details see the official LEGO Mindstorms communication protocol.

Set `RemoveFromRemote` to `true` to clear the just retrieved message from the NXT's mailbox (and free occupied memory). Set it to `false` to just "look into" the message while it will still remain on the NXT's message queue.

`message` contains the actual message (string) that has been retrieved. `localInboxReturn` is just the mailbox number that the message was read from (again, see official Mindstorms communication protocol).

Optionally, the packet's statusbyte is returned in the output argument `statusByte`, if requested. Warning from this functions will then be supressed (i.e. no warnings are raised then).

If no NXT handle is specified the default one (`COM_GetDefaultNXT`) is used.

Note:

This command can only be used when an external program (e.g. written in NXT-G, NXC or NBC) is running on the NXT. Otherwise a warning will be thrown (and an empty message will be returned).

Use this function to read data locally stored on the NXT. There are 10 usable mailbox queues, each with a certain size (so be careful to avoid overflows). Maximum message limit is 58 bytes / chars. This function can be used to communicate with NXC programs

(the NXC-function "SendMessage" can be used to write the data on the NXT).

Examples

```
NXT_MessageWrite('Test message', 0);
pause(1)
% an NXC program will process this message from inbox 0
% and generate / "send" an answer to inbox 1 for us
reply = NXT_MessageRead(1, 1, true);
```

See also

NXT_MessageWrite,

Signature

- **Author:** Linus Atorf (see AUTHORS)
- **Date:** 2009/08/31
- **Copyright:** 2007-2010, RWTH Aachen University

Published with wg_publish; V1.0

NXT_MessageWrite

Writes a "NXT-to-NXT message" to the NXT's incoming BT mailbox queue

Contents

- Syntax:
- Description:
- Examples:
- See also
- Signature

Syntax:

```
NXT_MessageWrite(message)

NXT_MessageWrite(message, mailbox)

NXT_MessageWrite(message, mailbox, handle)
```

Description:

`NXT_MessageWrite(message)` sends given `message` to the NXT brick

`NXT_MessageWrite(message, mailbox)` stores `message` in the specified `mailbox`. If no mailbox is specified, default one is 0 (zero)

`NXT_MessageWrite(message, mailbox, handle)` uses the given NXT connection `handle`. If no handle is specified, the default one (`COM_GetDefaultNXT()`) is used.

Note:

Use this function to store data locally on the NXT. There are 10 usable mailbox queues, each with a certain size (so be careful to avoid overflows). Maximum message limit is 58 bytes / chars. This function can be used to communicate with NXC programs (the NXC-function "ReceiveRemoteString" can be used to read the data on the NXT).

Examples:

```
NXT_MessageWrite('F010045');
```

```
NXT_MessageWrite('F010045', 1);
```

```
handle = COM_OpenNXT();
NXT_MessageWrite('F010045', 0, handle);
```

See also

NXT_MessageRead,

Signature

- **Author:** Laurent Vaylet, The MathWorks SAS (France), Alexander Behrens (see AUTHORS)
- **Date:** 2008/12/17
-

Published with wg_publish; V1.0

NXT_PlaySoundFile

Plays the given sound file on the NXT Brick

Contents

- Syntax
- Description
- Examples
- See also
- Signature

Syntax

```
NXT_PlaySoundFile(filename, 'loop')

NXT_PlaySoundFile(filename, '', handle)
```

Description

`NXT_PlaySoundFile(filename, loop)` plays the soundfile stored on NXT Brick determined by the string `filename`. The maximum length is limited to 15 characters. The file extension '.rso' is added automatically if it was omitted. If the loop parameter is equal to `'loop'` the playback loop is activated.

`NXT_PlaySoundFile(name, loop, handle)` uses the given NXT connection `handle`. This should be a a struct containing a serial handle on a PC system and a file handle on a Linux system.

If no Bluetooth `handle` is specified the default one (`COM_GetDefaultNXT`) is used.

For more details see the official LEGO Mindstorms communication protocol.

Examples

```
NXT_PlaySoundFile('Goodmorning', 0);
```

```
handle = NXT_OpenNXT('bluetooth.ini');
NXT_StartProgram('Goodmorning.rso', 1, handle);
```

See also

NXT_StopSoundPlayback,

Signature

- **Author:** Alexander Behrens (see AUTHORS)
- **Date:** 2008/05/22
- **Copyright:** 2007-2010, RWTH Aachen University

Published with wg_publish; V1.0

NXT_PlayTone

Plays a tone with the given frequency and duration

Contents

- Syntax
- Description
- Examples
- See also
- Signature

Syntax

```
NXT_PlayTone(frequency, duration)
```

```
NXT_PlayTone(frequency, duration, handle)
```

Description

`NXT_PlayTone(frequency, duration)` plays a tone of the `frequency` in Hz (200 - 14000Hz) and the `duration` in milli seconds.

`NXT_PlayTone(frequency, duration, handle)` sends the play tone command over the specific NXT handle (e.g. struct containing a serial handle (PC) / file handle (Linux)).

If no NXT handle is specified the default one (`COM_GetDefaultNXT`) is used.

For more details see the official LEGO Mindstorms communication protocol.

Examples

```
NXT_PlayTone(440, 100);
```

```
handle = COM_OpenNXT('bluetooth.ini');
COM_SetDefaultNXT(handle);
NXT_PlayTone(1200, 120);
```

See also

COM_GetDefaultNXT,

Signature

- **Author:** Linus Atorf (see AUTHORS)
- **Date:** 2007/10/15
- **Copyright:** 2007-2010, RWTH Aachen University

Published with wg_publish; V1.0

NXT_ReadIOMap

Reads the IO map of the given module ID

Contents

- Syntax
- Description
- Examples
- See also
- Signature

Syntax

```
bytes = NXT_ReadIOMap(mod_id, offset, n_bytes)

bytes = NXT_ReadIOMap(mod_id, offset, n_bytes, handle)
```

Description

`bytes = NXT_ReadIOMap(mod_id, offset, n_bytes)` returns the data `bytes` of the module identified by the given module ID `mod_id`. The total number of bytes is determined by `n_bytes` and the position of the first byte index by the `offset` parameter.

`bytes = NXT_ReadIOMap(mod_id, offset, n_bytes, handle)` sends the IO map read command over the specific NXT `handle` (e.g. serial handle (PC) / file handle (Linux)).

If no NXT `handle` is specified the default one (`COM_GetDefaultNXT`) is used.

For more details see the official LEGO Mindstorms communication protocol.

Examples

```
OutputModuleID = 131073
bytes = NXT_ReadIOMap(OutputModuleID, 0, 29);
```

```
handle = COM_OpenNXT('bluetooth.ini');
OutputModuleID = 131073
SoundModuleID = 524289, 0, 30, handle);
```

See also

NXT_WriteIOMap, COM_GetDefaultNXT,

Signature

- **Author:** Alexander Behrens (see AUTHORS)
- **Date:** 2008/05/22
- **Copyright:** 2007-2010, RWTH Aachen University

NXT_ResetInputScaledValue

Resets the sensor's ScaledVal back to 0 (depends on current sensor mode)

Contents

- Syntax
- Description
- Examples
- See also
- Signature

Syntax

```
NXT_ResetInputScaledValue(port)
```

```
NXT_ResetInputScaledValue(port, handle)
```

Description

`NXT_ResetInputScaledValue(port)` resets the sensors `ScaledVal` back to `0` of the given sensor `port`. The value `port` can be addressed by the symbolic constants `SENSOR_1`, `SENSOR_2`, `SENSOR_3` and `SENSOR_4` analog to the labeling on the NXT Brick. The `ScaledVal` is set by function `NXT_SetInputMode`.

`NXT_ResetInputScaledValue(port, handle)` uses the given NXT connection `handle`. This should be a struct containing a serial handle on a PC system and a file handle on a Linux system.

If no NXT handle is specified the default one (`COM_GetDefaultNXT`) is used.

For more details see the official LEGO Mindstorms communication protocol.

Note:

This function should be called after using `NXT_SetInputMode`, before you want to actually use your new special input value (to make sure counting starts at zero). See `NXT_GetInputValues` for more details about what kind of values are returned.

Examples

```
NXT_ResetInputScaledValue(SENSOR_2);
```

```
handle = COM_OpenNXT('bluetooth.ini');
NXT_ResetInputScaledValue(SENSOR_4, handle);
```

See also

NXT_SetInputMode, NXT_GetInputValues,

Signature

- **Author:** Linus Atorf (see AUTHORS)
- **Date:** 2007/10/15
- **Copyright:** 2007-2010, RWTH Aachen University

Published with wg_publish; V1.0

NXT_ResetMotorPosition

Resets NXT internal counter for specified motor, relative or absolute counter

Contents

- Syntax
- Description
- Examples
- See also
- Signature

Syntax

```
NXT_ResetMotorPosition(port, isRelative)
```

```
NXT_ResetMotorPosition(port, isRelative, handle)
```

Description

`NXT_ResetMotorPosition(port, isRelative)` resets the NXT internal counter of the given motor `port`. The value `port` can be addressed by the symbolic constants `MOTOR_A`, `MOTOR_B`, `MOTOR_C` analog to the labeling on the NXT Brick. The boolean flag `isRelative` determines the relative (BlockTachoCount) or absolute counter (RotationCount).

`NXT_ResetMotorPosition(port, handle)` uses the given NXT connection `handle`. This should be a struct containing a serial handle on a PC system and a file handle on a Linux system.

If no NXT handle is specified the default one (`COM_GetDefaultNXT`) is used.

For more details see the official LEGO Mindstorms communication protocol.

Examples

```
NXT_ResetMotorPosition(MOTOR_B, true);
```

```
handle = COM_OpenNXT('bluetooth.ini');
NXT_ResetMotorPosition(MOTOR_A, false, handle);
```

See also

NXTMotor, ResetPosition, NXC_ResetErrorCorrection,

Signature

- **Author:** Linus Atorf (see AUTHORS)
- **Date:** 2007/10/15
- **Copyright:** 2007-2010, RWTH Aachen University

Published with wg_publish; V1.0

NXT_SendKeepAlive

Sends a KeepAlive packet. Optional: requests sleep time limit.

Contents

- Syntax
- Description
- Examples
- See also
- Signature

Syntax

```
[status SleepTimeLimit] = NXT_SendKeepAlive(ReplyMode)
```

```
[status SleepTimeLimit] = NXT_SendKeepAlive(ReplyMode, handle)
```

Description

`[status SleepTimeLimit] = NXT_SendKeepAlive(ReplyMode)` sends a KeepAlive packet to the NXT Brick to get the current sleep time limit of the Brick in milliseconds. By the `ReplyMode` one can request an acknowledgement for the packet transmission. The two strings `'reply'` and `'dontreply'` are valid. `status` indicates if an error occures by the packet transmission. Function `checkStatusBytes` is interpreting this information per default. The value `SleepTimeLimit` contains the time in milliseconds after the NXT brick will turn off automatically. The variable will only be set if `ReplyMode` is `'reply'`. The sleep time limit setting can only be modified using the on-screen-menu on the brick itself.

Using `'dontreply'` will just send a keep-alive packet. This means, the NXT internal counter when to shut down automatically (this is a setting that can only be accessed directly on the NXT) will be reset. This counter is not an inactivity counter: Bluetooth traffic will NOT stop the NXT from turning off. E.g. if the sleep limit is set to 10 minutes, the only way to keep the NXT Brick from turning off is to send a keep-alive packet within this time.

If you use replymode `'reply'`, SleepTimeLimit tells you the current setting on the brick, in milliseconds. 0 means sleep timer is disabled. -1 is an invalid answer: You obviously didn't use `'reply'` and still tried to get an answer.

`[status SleepTimeLimit] = NXT_SendKeepAlive(ReplyMode, handle)` uses the given NXT connection `handle`. This should be a struct containing a serial handle on a PC system and a file handle on a Linux system.

If no NXT handle is specified the default one (`COM_GetDefaultNXT`) is used.

For more details see the official LEGO Mindstorms communication protocol.

Note:

This function is also called by `COM_OpenNXT()`. Then a keep-alive packet is send and the answer will be received to check for a correctly working bidirectional bluetooth connection.

Examples

```
[status SleepTimeLimit] = NXT_SendKeepAlive('reply');
```

```
NXT_SendKeepAlive('dontreply');
```

```
handle = COM_OpenNXT('bluetooth.ini');
[status SleepTimeLimit] = NXT_SendKeepAlive('reply', handle);
```

See also

COM_OpenNXT, **NXT_GetBatteryLevel**,

Signature

- **Author:** Linus Atorf (see AUTHORS)
- **Date:** 2007/10/15
- **Copyright:** 2007-2010, RWTH Aachen University

Published with wg_publish; V1.0

NXT_SetBrickName

Sets a new name for the NXT Brick (connected to the specified handle)

Contents

- Syntax
- Description
- Examples
- See also
- Signature

Syntax

```
[NXT_SetBrickName(name)

NXT_SetBrickName(name, handle)
```

Description

`NXT_SetBrickName(name)` sets a new `name` for the NXT Brick. The value `name` is a string value and determines the new name of the Brick. The maximum length is limited to 15 characters.

`NXT_SetBrickName(name, handle)` uses the given NXT connection `handle`. This should be a struct containing a serial handle on a PC system and a file handle on a Linux system.

If no NXT handle is specified the default one (`COM_GetDefaultNXT`) is used.

For more details see the official LEGO Mindstorms communication protocol.

Examples

```
NXT_SetBrickName('MyRobot');
```

```
handle = COM_OpenNXT('bluetooth.ini');
NXT_SetBrickName('Mindy', handle);
```

See also

COM_GetDefaultNXT, NXT_SendKeepAlive, NXT_GetBatteryLevel,

Signature

- **Author:** Linus Atorf, Alexander Behrens (see AUTHORS)
- **Date:** 2007/10/15
- **Copyright:** 2007-2010, RWTH Aachen University

Published with wg_publish; V1.0

NXT_SetInputMode

Sets a sensor mode, configures and initializes a sensor to be read out

Contents

- Syntax
- Description
- Examples
- See also
- Signature

Syntax

```
status = NXT_SetInputMode(port, SensorTypeDesc, SensorModeDesc, ReplyMode)
```

```
status = NXT_SetInputMode(port, SensorTypeDesc, SensorModeDesc, ReplyMode, handle)
```

Description

`status = NXT_SetInputMode(InputPort, SensorTypeDesc, SensorModeDesc, ReplyMode)` sets mode, configures and initializes the given sensor `port` to be ready to be read out. The value `port` can be addressed by the symbolic constants `SENSOR_1`, `SENSOR_2`, `SENSOR_3` and `SENSOR_4` analog to the labeling on the NXT Brick. The value `SensorTypeDesc` determines the sensor type. See all valid types below. `SensorModeDesc` represents the sensor mode. It specifies what mode the `.ScaledVal` from `NXT_GetInputValues` should be. Valid parameters see below. By the `ReplyMode` one can request an acknowledgement for the packet transmission. The two strings `'reply'` and `'dontreply'` are valid. The return value `status` indicates if an error occures by the packet transmission.

`status = NXT_SetInputMode(InputPort, SensorTypeDesc, SensorModeDesc, ReplyMode, handle)` uses the given NXT connection `handle`. This should be a struct containing a serial handle on a PC system and a file handle on a Linux system.

If no NXT handle is specified the default one (`COM_GetDefaultNXT`) is used.

Input:

`SensorTypeDesc`: Valid types are (all strings):

`NO_SENSOR` (nothing, use to close sensor port)

`SWITCH` (NXT touch sensor, "binary")

`LIGHT_ACTIVE` (NXT light sensor, red LED is on)

`LIGHT_INACTIVE` (NXT light sensor, red LED is off)

`SOUND_DB` (NXT sound sensor, unit dB)

`SOUND_DBA` (NXT sound sensor, unit dBA)

`LOWSPEED` (NXT, passive digital sensor)

LOWSPEED_9V (NXT, active digital sensor, e.g. UltraSonic)

HIGHSPEED (NXT, probably digital sensor on highspeed port 4)

TEMPERATURE (old RCX sensor)

REFLECTION (old RCX sensor)

ANGLE (old RCX sensor)

COLORFULL (NXT 2.0 Color sensor, full RGB mode)

COLORRED (NXT 2.0 Color sensor, red LED only)

COLORGREEN (NXT 2.0 Color sensor, green LED only)

COLORBLUE (NXT 2.0 Color sensor, blue LED only)

COLORNONE (NXT 2.0 Color sensor, no LED)

SensorModeDesc: Valid modes are (all strings):

RAWMODE (Fastest. RawADVal will be used)

BOOLEANMODE (1 if above 45% threshold, else 0)

TRANSITIONCNTMODE (count transitions of booleanmode)

PERIODCOUNTERMODE (count periods (up and down transition) of boolean mode)

PCTFULLSCALEMODE (normalized percentage between 0 and 100, use .NormalizedADVal instead!)

More exotic modes are: CELSIUSMODE (RCX temperature only)

FAHRENHEITMODE (RCX temperature only)

ANGLESTEPSMODE (RCX rotation only)

SLOPEMASK (what's this???)

MODEMASK (what's this???)

For more details see the official LEGO Mindstorms communication protocol.

Examples

```
status = NXT_SetInputMode(SENSOR_1, 'SOUND_DB', 'RAWMODE', 'dontreply');
```

```
handle = COM_OpenNXT('bluetooth.ini');
status = NXT_SetInputMode(SENSOR_3, 'LIGHT_ACTIVE', 'RAWMODE', 'dontreply', handle);
```

See also

NXT_GetInputValues, **OpenLight**, **OpenSound**, **OpenSwitch**, **OpenUltrasonic**, **CloseSensor**, **SENSOR_1**, **SENSOR_2**, **SENSOR_3**, **SENSOR_4**,

Signature

- **Author:** Linus Atorf (see AUTHORS)
- **Date:** 2007/10/15
- **Copyright:** 2007-2010, RWTH Aachen University

Published with wg_publish; V1.0

NXT_SetOutputState

Sends previously specified settings to current active motor.

Contents

- Syntax
- Description
- Example
- See also
- Signature

Syntax

```
status = NXT_SetOutputState(port, power, IsMotorOn, IsBrake, RegModeName, TurnRatio, RunStateName, TachoLimit, ReplyMode)

status = NXT_SetOutputState(port, power, IsMotorOn, IsBrake, RegModeName, TurnRatio, RunStateName, TachoLimit, ReplyMode, handle)
```

Description

`status = NXT_SetOutputState(OutputPort, Power, IsMotorOn, IsBrake, RegModeName, TurnRatio, RunStateName, TachoLimit, ReplyMode)` sends the given settings like motor `port` (`MOTOR_A`, `MOTOR_B` or `MOTOR_C`), the `power` (`-100...100`, the `IsMotorOn` boolean flag, the `IsBrake` boolean flag, the regulation mode name `RegModeName` (`'IDLE'`, `'SYNC'`, `'SPEED'`), the `TurnRatio` (`-100...100`), the `RunStateName` (`'IDLE'`, `'RUNNING'`, `'RAMUP'`, `'RAMPDOWN'`), the `TachoLimit` (angle limit) in degrees, and the `ReplyMode`. By the `ReplyMode` one can request an acknowledgement for the packet transmission. The two strings `'reply'` and `'dontreply'` are valid. The return value `status` indicates if an error occures by the packet transmission.

`status = NXT_SetOutputState(OutputPort, Power, IsMotorOn, IsBrake, RegModeName, TurnRatio, RunStateName, TachoLimit, ReplyMode, handle)` uses the given NXT connection `handle`. This should be a serial handle on a PC system and a file handle on a Linux system.

Example

```
NXT_SetOutputState(MOTOR_A, 80, true, true, 'SPEED', 0, 'RUNNING', 360, 'dontreply');
```

See also

DirectMotorCommand, NXT_GetOutputState, ReadFromNXT, MOTOR_A, MOTOR_B, MOTOR_C,

Signature

- **Author:** Linus Atorf (see AUTHORS)
- **Date:** 2007/10/15
- **Copyright:** 2007-2010, RWTH Aachen University

Published with wg_publish; V1.0

NXT_StartProgram

Starts the given program on the NXT Brick

Contents

- Syntax
- Description
- Examples
- See also
- Signature

Syntax

```
NXT_StartProgram(filename, [handle])

status = NXT_StartProgram(filename, [handle])
```

Description

`NXT_StartProgram(filename)` starts the embedded NXT Brick program determined by the string `filename`. The maximum length is limited to 15 characters. The file extension `'.rxe'` is added automatically if it was omitted. The output argument `status` is optional and return the error status of the collected packet. If it is omitted, not reply packet will be requested (this is significantly faster via Bluetooth).

The last parameter `handle` is optional. If no NXT handle is specified the default one (`COM_GetDefaultNXT`) is used.

For more details see the official LEGO Mindstorms communication protocol.

Examples

```
NXT_StartProgram('ResetCounter');
```

```
handle = COM_OpenNXT('bluetooth.ini');
NXT_StartProgram('Demo.rxe', handle);
```

See also

NXT_StopProgram, NXT_GetCurrentProgramName,

Signature

- **Author:** Alexander Behrens, Linus Atorf (see AUTHORS)
- **Date:** 2007/10/15
- **Copyright:** 2007-2010, RWTH Aachen University

Published with wg_publish; V1.0

NXT_StopProgram

Stops the currently running program on the NXT Brick

Contents

- Syntax
- Description
- Examples
- See also
- Signature

Syntax

`NXT_StopProgram()`

`NXT_StopProgram(handle)`

Description

`NXT_StopProgram()` stops the current running embedded NXT Brick program.

`NXT_StopProgram(handle)` uses the given NXT connection `handle`. This should be a struct containing a serial handle on a PC system and a file handle on a Linux system.

If no Bluetooth handle is specified the default one (`COM_GetDefaultNXT`) is used.

For more details see the official LEGO Mindstorms communication protocol.

Examples

```
NXT_StopProgram();
```

```
handle = COM_OpenNXT('bluetooth.ini');
NXT_StopProgram(handle);
```

See also

NXT_StartProgram, NXT_GetCurrentProgramName,

Signature

- **Author:** Alexander Behrens, Linus Atorf (see AUTHORS)
- **Date:** 2007/10/15
- **Copyright:** 2007-2010, RWTH Aachen University

Published with wg_publish; V1.0

NXT_StopSoundPlayback

Stops the current sound playback

Contents

- Syntax
- Description
- Examples
- See also
- Signature

Syntax

```
NXT_StopSoundPlayback()

NXT_StopSoundPlayback(handle)
```

Description

`NXT_StopSoundPlayback()` stops the current sound playback.

`NXT_StopSoundPlayback(handle)` sends the stop sound playback command over the specific Bluetooth `handle` (serial handle (PC) / file handle (Linux)).

If no NXT `handle` is specified the default one (`COM_GetDefaultNXT`) is used.

For more details see the official LEGO Mindstorms communication protocol.

Examples

```
NXT_StopSoundPlayback();
```

```
handle = COM_OpenNXT('bluetooth.ini');
NXT_StopSoundPlayback(handle);
```

See also

NXT_PlaySoundFile, NXT_PlayTone, COM_GetDefaultNXT,

Signature

- **Author:** Alexander Behrens (see AUTHORS)
- **Date:** 2008/05/22
- **Copyright:** 2007-2010, RWTH Aachen University

Published with wg_publish; V1.0

NXT_WriteIOMap

Writes the IO map to the given module ID

Contents

- Syntax
- Description
- Examples
- See also
- Signature

Syntax

```
NXT_WriteIOMap(mod_id, offset, n_bytes, data)
```

```
NXT_WriteIOMap(mod_id, offset, n_bytes, data, handle)
```

Description

`NXT_WriteIOMap(mod_id, offset, n_bytes, data)` write the data `bytes` to the given module ID (`mod_id`). The total number of bytes is given by `n_bytes`. The `offset` parameter determines the index position of the first byte.

`NXT_WriteIOMap(mod_id, offset, n_bytes, data, handle)` sends the IO map write command over the specific NXT handle `handle` (e.g. serial handle (PC) / file handle (Linux)).

If no NXT `handle` is specified the default one (`COM_GetDefaultNXT`) is used.

For more details see the official LEGO Mindstorms communication protocol.

Examples

```
OutputModuleID = 131073
NXT_WriteIOMap(OutputModuleID, 0, 30, bytes;
```

```
handle = COM_OpenNXT('bluetooth.ini');
OutputModuleID = 131073
NXT_WriteIOMap(OutputModuleID, 0, 30, bytes, handle);
```

See also

NXT_ReadIOMap, COM_GetDefaultNXT,

Signature

- **Author:** Alexander Behrens (see AUTHORS)
- **Date:** 2008/05/22
- **Copyright:** 2007-2010, RWTH Aachen University

OpenAccelerator

Initializes the HiTechnic acceleration sensor, sets correct sensor mode

Contents

- Syntax
- Description
- Examples
- See also
- Signature

Syntax

```
OpenAccelerator(port)

OpenAccelerator(port, handle)
```

Description

`OpenAccelerator(port)` initializes the input mode of NXT accelerator sensor specified by the sensor `port`. The value `port` can be addressed by the symbolic constants `SENSOR_1` , `SENSOR_2`, `SENSOR_3` and `SENSOR_4` analog to the labeling on the NXT Brick.

The last optional argument can be a valid NXT handle. If none is specified, the default handle will be used (call `COM_SetDefaultNXT` to set one).

Examples

```
OpenAccelerator(SENSOR_4);
acc_Vector = GetAccelerator(SENSOR_4);
CloseSensor(SENSOR_4);
```

See also

GetAccelerator, CloseSensor, COM_ReadI2C, NXT_LSGetStatus, NXT_LSRead,

Signature

- **Author:** Linus Atorf (see AUTHORS)
- **Date:** 2008/09/25
- **Copyright:** 2007-2010, RWTH Aachen University

Published with wg_publish; V1.0

OpenColor

Initializes the HiTechnic color V1 or V2 sensor, sets correct sensor mode

Contents

- Syntax
- Description
- Limitations
- Examples
- See also
- Signature

Syntax

`OpenColor(port)`

`OpenColor(port, handle)`

Description

`OpenColor(port)` initializes the input mode of HiTechnic Color sensor specified by the sensor port. The value `port` can be addressed by the symbolic constants `SENSOR_1` , `SENSOR_2`, `SENSOR_3` and `SENSOR_4` analog to the labeling on the NXT Brick. This function works for both HiTechnic Color sensors V1 and V2.

With `GetColor(port)` you can receive color values as RGB or Index.

The last optional argument can be a valid NXT handle. If none is specified, the default handle will be used (call `COM_SetDefaultNXT` to set one).

Since the Color sensor is a digital sensor (that uses the I²C protocol), the function `NXT_SetInputMode` cannot be used as for analog sensors.

Limitations

It's by design that the white LED of the Color sensors cannot be turned off by calling `CloseSensor`. It's always on when the sensor is connected. The V2 hardware version of the sensor performs significantly better than the V1 version.

Examples

```
OpenColor(SENSOR_2);
[index r g b] = GetColor(SENSOR_2, 0);
CloseSensor(SENSOR_2);
```

See also

GetColor, CalibrateColor, CloseSensor, OpenNXT2Color, GetNXT2Color, COM_ReadI2C,

Signature

- **Author:** Rainer Schnitzler, Linus Atorf (see AUTHORS)
- **Date:** 2010/09/16
- **Copyright:** 2007-2010, RWTH Aachen University

Published with wg_publish; V1.0

OpenCompass

Initializes the HiTechnic magnetic compass sensor, sets correct sensor mode

Contents

- Syntax
- Description
- Examples
- See also
- Signature

Syntax

```
OpenCompass(port)

OpenCompass(port, handle)
```

Description

`OpenCompass(port)` initializes the input mode of HiTechnic compass sensor specified by the sensor port. The value `port` can be addressed by the symbolic constants `SENSOR_1` , `SENSOR_2`, `SENSOR_3` and `SENSOR_4` analog to the labeling on the NXT Brick.

With `GetCompass(port)` you can receive the heading value ranging from 0 to 359.

The last optional argument can be a valid NXT handle. If none is specified, the default handle will be used (call `COM_SetDefaultNXT` to set one).

Since the compass sensor is a digital sensor (that uses the I²C protocol), the function `NXT_SetInputMode` cannot be used as for analog sensors.

Examples

```
OpenCompass(SENSOR_2);
degree = GetCompass(SENSOR_2);
CloseSensor(SENSOR_2);
```

See also

GetCompass, CloseSensor, COM_ReadI2C, NXT_LSGetStatus, NXT_LSRead,

Signature

- **Author:** Rainer Schnitzler (see AUTHORS)
- **Date:** 2008/08/01
- **Copyright:** 2007-2010, RWTH Aachen University

Published with wg_publish; V1.0

OpenEOPD

Initializes the HiTechnic EOPD sensor, sets correct sensor mode

Contents

- Syntax
- Description
- Limitations
- Example
- See also
- Signature

Syntax

```
OpenEOPD(port, range)

OpenEOPD(port, range, handle)
```

Description

`OpenEOPD(port, range)` initializes the HiTechnic EOPD sensor on the specified sensor port. This sensor can be used to accurately detect objects and small changes in distance to a target. It works by measuring the light returned from its own light source, so it can also be used to detect the "shinyness" and color of a surface.

`range` can be set to either `'SHORT'`, which covers a range of about 10cm, or it can be set to `'LONG'`, which enables increased sensitivity for up to 20cm.

The value `port` can be addressed by the symbolic constants `SENSOR_1` , `SENSOR_2`, `SENSOR_3` and `SENSOR_4` analog to the labeling on the NXT Brick.

Before the sensor can be used, `CalibrateEOPD` should be called, otherwise only raw values will be usable. Values can be retrieved using `GetEOPD`.

The last optional argument can be a valid NXT handle. If none is specified, the default handle will be used (call `COM_SetDefaultNXT` to set one).

Note:

For more details on calibration, see help text and examples of `CalibrateEOPD`.

Since each EOPD sensor uses a slightly different pulse frequency for the LED, multiple sensor can be used at once without influencing each other.

Limitations

It is normal that the red LED always stays on once the sensor is connected. The LED cannot be turned off using `CloseSensor`.

Example

```
port = SENSOR_2;
OpenEOPD(port, 'SHORT');

% set calibration matrix
calibMatrix = [3 91; 9 19];
CalibrateEOPD(port, 'SETMATRIX', calibMatrix);

% now the sensor can be used
[dist raw] = GetEOPD(port);

% clean up, as usual. LED stays on anyway
CloseSensor(port);
```

See also

CalibrateEOPD, **GetEOPD**, **CloseSensor**, **NXT_SetInputMode**, **NXT_GetInputValues**,

Signature

- **Author:** Linus Atorf (see AUTHORS)
- **Date:** 2009/09/17
- **Copyright:** 2007-2010, RWTH Aachen University

Published with wg_publish; V1.0

OpenGyro

Initializes the HiTechnic Gyroscopic sensor, sets correct sensor mode

Contents

- Syntax
- Description
- Examples
- See also
- Signature

Syntax

```
OpenGyro(port)

OpenGyro(port, handle)
```

Description

`OpenGyro(port)` initializes the HiTechnic Gyro sensor on the specified sensor port. The value `port` can be addressed by the symbolic constants `SENSOR_1` , `SENSOR_2`, `SENSOR_3` and `SENSOR_4` analog to the labeling on the NXT Brick.

Before the sensor can be used, `CalibrateGyro` has to be called.

With `GetGyro(port)` you can receive Gyro values up to a max. of +/- 360° rotation rate per sec.

The last optional argument can be a valid NXT handle. If none is specified, the default handle will be used (call `COM_SetDefaultNXT` to set one).

Note:

For more details on calibration, see help text and examples of `CalibrateGyro`.

Examples

```
OpenGyro(SENSOR_2);
CalibrateGyro(SENSOR_2, 'AUTO');
speed = GetGyro(SENSOR_2);
CloseSensor(SENSOR_2);
```

See also

CalibrateGyro, GetGyro, CloseSensor, NXT_SetInputMode, NXT_GetInputValues,

Signature

- **Author:** Rainer Schnitzler, Linus Atorf (see AUTHORS)
- **Date:** 2009/08/31
- **Copyright:** 2007-2010, RWTH Aachen University

OpenInfrared

Initializes the HiTechnic infrared seeker sensor, sets correct sensor mode

Contents

- Syntax
- Description
- Examples
- See also
- Signature

Syntax

```
OpenInfrared(port)

OpenInfrared(port, handle)
```

Description

`OpenInfrared(port)` initializes the input mode of HiTechnic infrared seeker sensor specified by the sensor `port`. The value `port` can be addressed by the symbolic constants `SENSOR_1` , `SENSOR_2`, `SENSOR_3` and `SENSOR_4` analog to the labeling on the NXT Brick.

The last optional argument can be a valid NXT handle. If none is specified, the default handle will be used (call `COM_SetDefaultNXT` to set one).

Examples

```
OpenInfrared(SENSOR_4);
[direction rawData] = GetInfrared(SENSOR_4);
CloseSensor(SENSOR_4);
```

See also

GetInfrared, CloseSensor, NXT_LSGetStatus, NXT_LSRead,

Signature

- **Author:** Linus Atorf (see AUTHORS)
- **Date:** 2008/09/25
- **Copyright:** 2007-2010, RWTH Aachen University

Published with wg_publish; V1.0

OpenLight

Initializes the NXT light sensor, sets correct sensor mode

Contents

- Syntax
- Description
- Example
- See also
- Signature

Syntax

```
OpenLight(port, mode)

OpenLight(port, mode, handle)
```

Description

`OpenLight(port, mode)` initializes the input mode of NXT light sensor specified by the sensor `port` and the light `mode`. The value `port` can be addressed by the symbolic constants `SENSOR_1` , `SENSOR_2`, `SENSOR_3` and `SENSOR_4` analog to the labeling on the NXT Brick. The `mode` represents one of two modes `'ACTIVE'` (active illumination: red light on) and `'INACTIVE'` (passive illumination red light off). To deactive the active illumination the function `CloseSensor` is used.

The last optional argument can be a valid NXT handle. If none is specified, the default handle will be used (call `COM_SetDefaultNXT` to set one).

For more complex settings the function `NXT_SetInputMode` can be used.

Example

```
OpenLight(SENSOR_1, 'ACTIVE');
light = GetLight(SENSOR_1);
CloseSensor(SENSOR_1);
```

See also

CloseSensor, GetLight, OpenNXT2Color, GetNXT2Color, NXT_SetInputMode, SENSOR_1, SENSOR_2, SENSOR_3, SENSOR_4,

Signature

- **Author:** Linus Atorf, Alexander Behrens (see AUTHORS)
- **Date:** 2007/10/15
- **Copyright:** 2007-2010, RWTH Aachen University

Published with wg_publish; V1.0

OpenNXT2Color

Initializes the LEGO color sensor from the NXT 2.0 set, sets correct sensor mode

Contents

- Syntax
- Description
- Limitations
- Examples
- See also
- Signature

Syntax

```
OpenNXT2Color(port, mode)
```

```
OpenNXT2Color(port, mode, handle)
```

Description

This function initializes the input mode of the LEGO NXT 2.0 Color sensor specified by the sensor port. The value `port` can be addressed by the symbolic constants `SENSOR_1` , `SENSOR_2`, `SENSOR_3` and `SENSOR_4` analog to the labeling on the NXT Brick. This function is intended for the Color sensor that comes with the NXT 2.0 set. It has the label "RGB" written on the front, 3 LED openings (1 black empty spot, the light sensor and a clear lens with tiny red, green, blue LEDs behind it). It is not to be confused with the HiTechnic Color sensors (V1 and V2), for those please see the functions `OpenColor` and `GetColor`.

With `GetNXT2Color(port)` you can receive the detected brightness or color.

`mode` specifies the operating mode of the sensor, the following values are allowed:

- `'FULL'` - The red, green, and blue LEDs are constantly on (actually flashing at a high frequency), and the sensor will try to detect one of 6 predefined colors.
- `'RED'` - The red LED is constantly on, the sensor outputs reflected light / brightness. This is similar to the LEGO Light sensor mode `'ACTIVE'`. See `OpenLight`.
- `'GREEN'` - The green LED is constantly on, the sensor outputs reflected light / brightness.
- `'BLUE'` - The blue LED is constantly on, the sensor outputs reflected light / brightness.
- `'NONE'` - All LEDs are constantly off, the sensor outputs ambient light / brightness. This is similar to the LEGO Light sensor mode `'INACTIVE'`. See `OpenLight`.

The last optional argument can be a valid NXT handle. If none is specified, the default handle will be used (call `COM_SetDefaultNXT` to set one).

Limitations

The sensor is influenced by ambient light. It reacts differently on daylight than on artificial

light. The modes 'RED' and 'NONE' are similar to the Light sensor's modes 'ACTIVE' and 'INACTIVE', but the sensors are not perfectly compatible.

Examples

```
port = SENSOR_1;
OpenNXT2Color(port, 'FULL');
color = GetNXT2Color(port);
if strcmp(color, 'BLUE')
    disp('Blue like the ocean');
else
    disp(['The detected color is ' color]);
end%if
CloseSensor(port);
```

```
port = SENSOR_2;
OpenNXT2Color(port, 'NONE');
colorVal = GetNXT2Color(port);
if colorVal > 700
    disp('It''s quite bright!')
end%if
CloseSensor(port);
```

See also

GetNXT2Color, **CloseSensor**, **OpenColor**, **GetColor**, **OpenLight**, **GetLight**, **COM_ReadI2C**,

Signature

- **Author:** Nick Watts, Linus Atorf (see AUTHORS)
- **Date:** 2010/09/21
- **Copyright:** 2007-2010, RWTH Aachen University

Published with wg_publish; V1.0

OpenRFID

Initializes the Codatex RFID sensor, sets correct sensor mode

Contents

- Syntax
- Description
- Example
- See also
- Signature

Syntax

```
status = OpenRFID(port)
```

Description

`OpenRFID(port)` initializes the input mode of Codatex RFID sensor specified by the sensor port. The value port can be addressed by the symbolic constants `SENSOR_1`, `SENSOR_2`, `SENSOR_3` and `SENSOR_4` analog to the labeling on the NXT Brick.

The RF ID Sensor works with 125 kHz transponders compatible with EM4102 modulation type. With this sensor you can read 5-byte-long transponder numbers into the NXT. Since the NXT RFID sensor is a digital sensor (that uses the I²C protocol), the function `NXT_SetInputMode` cannot be used as for analog sensors.

Note:

Opening the sensor can take a while. Via Bluetooth, delays of more than 1 second are not uncommon.

Example

```
OpenRFID(SENSOR_2);
transID = GetRFID(SENSOR_2,'single');
CloseSensor(SENSOR_2);
```

See also

GetRFID, CloseSensor,

Signature

- **Author:** Linus Atorf, Rainer Schnitzler (see AUTHORS)
- **Date:** 2008/12/1
- **Copyright:** 2007-2010, RWTH Aachen University

Published with wg_publish; V1.0

OpenSound

Initializes the NXT sound sensor, sets correct sensor mode

Contents

- Syntax
- Description
- Examples
- See also
- Signature

Syntax

```
OpenSound(port, mode)

OpenSound(port, mode, handle)
```

Description

`OpenSound(port, mode)` initializes the input mode of NXT sound sensor specified by the sensor `port` and the sound `mode`. The value `port` can be addressed by the symbolic constants `SENSOR_1` , `SENSOR_2`, `SENSOR_3` and `SENSOR_4` analog to the labeling on the NXT Brick. The `mode` represents one of two modes `'DB'` (dB measurement) and `'DBA'` (dBA measurement)

The last optional argument can be a valid NXT handle. If none is specified, the default handle will be used (call `COM_SetDefaultNXT` to set one).

For more complex settings the function `NXT_SetInputMode` can be used.

Examples

```
OpenSound(SENSOR_1, 'DB');
sound = GetSound(SENSOR_1);
CloseSensor(SENSOR_1);
```

See also

NXT_SetInputMode, GetSound, CloseSensor, SENSOR_1, SENSOR_2, SENSOR_3, SENSOR_4,

Signature

- **Author:** Linus Atorf, Alexander Behrens (see AUTHORS)
- **Date:** 2010/09/14
- **Copyright:** 2007-2010, RWTH Aachen University

Published with wg_publish; V1.0

OpenSwitch

Initializes the NXT touch sensor, sets correct sensor mode

Contents

- Syntax
- Description
- Example
- See also
- Signature

Syntax

```
OpenSwitch(port)

OpenSwitch(port, handle)
```

Description

`OpenSound(port)` initializes the input mode of NXT switch / touch sensor specified by the sensor `port`. The value `port` can be addressed by the symbolic constants `SENSOR_1` , `SENSOR_2`, `SENSOR_3` and `SENSOR_4` analog to the labeling on the NXT Brick.

The last optional argument can be a valid NXT handle. If none is specified, the default handle will be used (call `COM_SetDefaultNXT` to set one).

For more complex settings the function `NXT_SetInputMode` can be used.

Example

```
OpenSwitch(SENSOR_4);
switchState = GetSwitch(SENSOR_4);
CloseSensor(SENSOR_4);
```

See also

NXT_SetInputMode, GetSwitch, CloseSensor, SENSOR_1, SENSOR_2, SENSOR_3, SENSOR_4,

Signature

- **Author:** Linus Atorf, Alexander Behrens (see AUTHORS)
- **Date:** 2007/10/15
- **Copyright:** 2007-2010, RWTH Aachen University

Published with wg_publish; V1.0

OpenUltrasonic

Initializes the NXT ultrasonic sensor, sets correct sensor mode

Contents

- Syntax
- Description
- Limitations
- Examples
- See also
- Signature

Syntax

```
OpenUltrasonic(port)

OpenUltrasonic(port, mode)

OpenUltrasonic(port, mode, handle)
```

Description

`OpenUltrasonic(port)` initializes the input mode of NXT ultrasonic sensor specified by the sensor `port`. The value `port` can be addressed by the symbolic constants `SENSOR_1` , `SENSOR_2`, `SENSOR_3` and `SENSOR_4` analog to the labeling on the NXT Brick.

`OpenUltrasonic(port, mode)` can enable the snapshot mode if the value `mode` is equal to the string `'snapshot'`. This mode provides the snap shot mode (or SINGLE_SHOT mode) of the NXT ultrasonic sensor, which provides several sensor readings in one step. See `USMakeSnapshot` for more information.

The last optional argument can be a valid NXT handle. If none is specified, the default handle will be used (call `COM_SetDefaultNXT` to set one).

Since the NXT ultrasonic sensor is a digital sensor (that uses the I²C protocol), the function `NXT_SetInputMode` cannot be used as for analog sensors.

Note:

When the US sensor is opened in snapshot mode, the function `GetUltrasonic` does not work correctly!

Limitations

Since the Ultrasonic sensors all operate at the same frequency, multiple US sensors will interfere with each other! If multiple US sensors can "see each other" (or their echos and reflections), results will be unpredictable (and probably also unusable). You can avoid this problem by turning off US sensors, or operating them in snapshot mode (see also `USMakeSnapshot` and `USGetSnapshotResults`).

Examples

```
OpenUltrasonic(SENSOR_4);
distance = GetUltrasonic(SENSOR_4);
CloseSensor(SENSOR_4);
```

```
port = SENSOR_4;
OpenUltrasonic(port, 'snapshot');
% send out the ping
USMakeSnapshot(port);
% wait some time for the sound to travel
pause(0.1); % 100ms is probably too much, calculate using c_sound ;-)
% retrieve all the echos in 1 step
echos = USGetSnapshotResults(port);
CloseSensor(SENSOR_4);
```

See also

GetUltrasonic, **USMakeSnapshot**, **USGetSnapshotResults**, **CloseSensor**,

Signature

- **Author:** Linus Atorf, Alexander Behrens (see AUTHORS)
- **Date:** 2008/01/08
- **Copyright:** 2007-2010, RWTH Aachen University

Published with wg_publish; V1.0

OptimizeToolboxPerformance

Copies binary versions of typecastc to toolbox for better performance

Contents

- Syntax
- Description
- Example
- Signature

Syntax

```
OptimizeToolboxPerformance
```

Description

This script automates toolbox optimization. The private functions wordbytes2dec and dec2wordbytes are very frequently called and thus most critical for performance (mostly, but not only CPU load), especially at high packet rates (USB). Profiling shows the bottleneck: typecast. When using the binary version from the MATLAB directory, speed-ups up to 300% can be experienced. However this binary version is in a private directory, so cannot be called directly. Also it depends on the OS, CPU architecture and MATLAB version used. So, the only solution: We copy those binary files to our own toolboxes private directory.

The script asks the user for permission and performs the file actions automatically. After each step the results will be checked to avoid a corrupt toolbox.

During the process, the m-files for the above named functions will be overwritten. This is the reason we provide 2 versions in the private dir, .m.slow (the original with typecast) and .m.fast (optimized version that needs binary typecastc files in the same directory).

Note: The optimization is not possible anymore since MATLAB Release 2010a (i.e. since MATLAB version 7.10 and greater). As `typecast` is now a built-in function, optimization is probably not necessary anymore anyway.

Example

```
OptimizeToolboxPerformance
```

Signature

- **Author:** Linus Atorf (see AUTHORS)
- **Date:** 2010/10/01
- **Copyright:** 2007-2010, RWTH Aachen University

Published with wg_publish; V1.0

ReadFromNXT

Reads current state of specified motor(s) from NXT brick

Contents

- Syntax
- Description
- Limitations:
- Examples:
- See also
- Signature

Syntax

```
DATA = OBJ.ReadFromNXT

DATA = OBJ.ReadFromNXT(HANDLE)

[DATA1 DATA2] = OBJ.ReadFromNXT

[DATA1 DATA2] = OBJ.ReadFromNXT(HANDLE)
```

Description

Request the current state of the motor object `OBJ` from the NXT brick. NXTMotor object `OBJ` is not modified. `DATA` is a structure with property/value pairs.

If the NXTMotor object `OBJ` controls two motors, `DATA1` and `DATA2` hold the parameters of the first and the second motor respectively. If only one output argument is given, the parameters of the first motor are returned.

Use the optional parameter `HANDLE` to identifiy the connection to use for this command. Otherwise the default handle (see `COM_SetDefaultNXT`) will be used.

The returned struct contains the following fields:

- `Port` - The motor port these data apply to.
- `Power` - The currently set power level (from -100 to 100).
- `Position` - The motor's position in degrees (of the internal rotation sensor). These encoders should be accurate to +/- 1 degree. Values will increase positively during forward motion, and decrease during reverse motion. You can reset this counter by using the method `.ResetPosition`.
- `IsRunning` - A boolean indicating whether the motor is currently spinning or actively braking (basically whether the motor *"is turned on or off"*). It will only be false once the motor is in free-running "coast" mode, i.e. when the power to this motor is turned off (e.g. after calling `Stop('off')` or when the set `TachoLimit` was reached). To clarify: if a motor was stopped using `.Stop('brake')`, or if `.ActionAtTachoLimit` was set to `'HoldBrake'`, `IsRunning` will keep returning true! Use the method `.WaitFor` to check wether a motor is ready to accept new commands!

`SpeedRegulation` - A boolean indicating wether the motor currently uses speed regulation. This is turned off during synchronous driving (when driving with 2 motors at the same time).

- `TachoLimit` - The currently set goal for the motor to reach. If set to 0, the motor is spinning forever.
- `TachoCount` - This counter indicates the progress of the motor for its current goal -- i.e. if a `TachoLimit ~= 0` is set, `TachoCount` will count up to this value during movement.

Note:

The values of `TachoCount` and `TachoLimit` (which can nicely be used as progress indicators) are not guaranteed to keep being valid once motor movement has finished. They can/will be cleared by the next `SendToNXT`, `Stop`, `StopMotor` or maybe even `DirectMotorCommand`.

For advanced users: The field `Position` maps to the NXT firmware's register / IOmap counter `RotationCount`, and `TachoCount` maps to `TachoCount` as expected.

Limitations:

Apart from the previously mentioned limitation of `IsRunning` (return values slightly differ from what would be expected), the value of `SpeedRegulation` can sometimes be unexpected: The `DATA` struct returned by `ReadFromNXT` always returns the true state of the NXT's firmware registers (it's basically just a wrapper for `NXT_GetOutputState`). When using an `NXTMotor` object with `SpeedRegulation = true`, the regulation will only be enabled during driving. When the motor control starts braking, regulation will be disabled, and this is what `ReadFromNXT` shows you. So don't worry when you receive `SpeedRegulation = false` using this method, even though you clearly enabled speed regulation. This is by design, and the motor did in fact use speed regulation during its movement.

Examples:

```
% Move motor A and show its state after 3 seconds
motorA = NXTMotor('A', 'Power', 50);
motorA.SendToNXT();
pause(3);
data = motorA.ReadFromNXT();
```

```
% Construct a NXTMotor object on port 'A' with a power of
% 50, TachoLimit of 1000, and send the motor settings to the NXT.
% Show the progress of the motor movement "on the fly".

motorA = NXTMotor('A', 'Power', 50, 'TachoLimit', 1000);
% this example wouldn't work with 'HoldBrake'
motorA.ActionAtTachoLimit = 'Brake';

motorA.SendToNXT();

% monitor during movement
data = motorA.ReadFromNXT();
while(data.IsRunning)
    percDone = abs(data.TachoCount / data.TachoLimit) * 100;
    disp(sprintf('Motor movement is %d % complete/n', percDone));
    data = motorA.ReadFromNXT(); % refresh
end%while
```

See also

NXTMotor, **SendToNXT**, **ResetPosition**, **NXT_GetOutputState**,

Signature

- **Author:** Aulis Telle, Linus Atorf (see AUTHORS)
- **Date:** 2008/11/12
- **Copyright:** 2007-2010, RWTH Aachen University

Published with wg_publish; V1.0

ResetPosition

Resets the position counter of the given motor(s).

Contents

- Syntax
- Description
- Example:
- See also
- Signature

Syntax

`OBJ.ResetPosition()`

`OBJ.ResetPosition(HANDLE)`

Description

Reset the position of the motor specified in `OBJ`. Only the internal states of the NXT brick corresponding to the specified motor(s) are reset. The motor is not moved. This resets the field `.Position` you can read out using `.ReadFromNXT`.

Specify `HANDLE` (optional) to identifiy the connection to use for this command. Otherwise the defaul handle (set using `COM_SetDefaultNXT`) will be used.

Note:

For advanced users: The field `Position` maps to the NXT firmware's register / IOmap counter `RotationCount`. It can also be reset using `NXT_ResetMotorPosition(port, false)`. That's in fact what this method does.

Example:

```
motorC = NXTMotor('C', 'Power', -20, 'TachoLimit', 120);
motorC.SendToNXT();
motorC.WaitFor();
data = motorC.ReadFromNXT()
%>> data =
%             Port: 2
%            Power: 0
%        IsRunning: 0
% SpeedRegulation: 0
%       TachoLimit: 120
%       TachoCount: -120
%         Position: -120

motorC.ResetPosition();
data = motorC.ReadFromNXT()
%>> data =
%             Port: 2
%            Power: 0
%        IsRunning: 0
% SpeedRegulation: 0
%       TachoLimit: 120
%       TachoCount: -120
%         Position: 0
```

See also

NXTMotor, **ReadFromNXT**, **NXT_ResetMotorPosition**, **NXT_GetOutputState**, **NXC_ResetErrorCorrection**,

Signature

- **Author:** Aulis Telle, Linus Atorf (see AUTHORS)
- **Date:** 2009/08/25
- **Copyright:** 2007-2010, RWTH Aachen University

Published with wg_publish; V1.0

SENSOR_1

Symbolic constant SENSOR_1 (returns 0)

Contents

- Syntax
- Description
- Example
- See also
- Signature

Syntax

```
port1 = SENSOR_1()
```

Description

`port1 = SENSOR_1()` returns `0` as the value `port1`.

Example

```
port1 = SENSOR_1()
%result: >> port1 = 0
```

See also

SENSOR_2, SENSOR_3, SENSOR_4,

Signature

- **Author:** Linus Atorf, Alexander Behrens (see AUTHORS)
- **Date:** 2007/10/15
- **Copyright:** 2007-2010, RWTH Aachen University

Published with wg_publish; V1.0

SENSOR_2

Symbolic constant SENSOR_2 (returns 1)

Contents

- Syntax
- Description
- Example
- See also
- Signature

Syntax

```
port2 = SENSOR_2()
```

Description

`port2 = SENSOR_2()` returns `1` as the value `port2`.

Example

```
port2 = SENSOR_2()
%result: >> port2 = 1
```

See also

SENSOR_1, SENSOR_3, SENSOR_4,

Signature

- **Author:** Linus Atorf, Alexander Behrens (see AUTHORS)
- **Date:** 2007/10/15
- **Copyright:** 2007-2010, RWTH Aachen University

Published with wg_publish; V1.0

SENSOR_3

Symbolic constant SENSOR_3 (returns 2)

Contents

- Syntax
- Description
- Example
- See also
- Signature

Syntax

```
port3 = SENSOR_3()
```

Description

`port3 = SENSOR_3()` returns `2` as the value `port3`.

Example

```
port3 = SENSOR_3()
%result: >> port3 = 2
```

See also

SENSOR_1, SENSOR_2, SENSOR_4,

Signature

- **Author:** Linus Atorf, Alexander Behrens (see AUTHORS)
- **Date:** 2007/10/15
- **Copyright:** 2007-2010, RWTH Aachen University

Published with wg_publish; V1.0

SENSOR_4

Symbolic constant SENSOR_4 (returns 3)

Contents

- Syntax
- Description
- Example
- See also
- Signature

Syntax

```
port4 = SENSOR_4()
```

Description

`port4 = SENSOR_4()` returns `3` as the value `port4`.

Example

```
port4 = SENSOR_4()
%result: >> port4 = 3
```

See also

SENSOR_1, SENSOR_2, SENSOR_3,

Signature

- **Author:** Linus Atorf, Alexander Behrens (see AUTHORS)
- **Date:** 2007/10/15
- **Copyright:** 2007-2010, RWTH Aachen University

Published with wg_publish; V1.0

SendToNXT

Send motor settings to the NXT brick

Contents

- Syntax
- Description
- Limitations
- Example
- See also
- Signature

Syntax

```
OBJ.SendToNXT

OBJ.SendToNXT(HANDLE)
```

Description

`OBJ.SendToNXT` sends the motor settings in `OBJ` to the NXT brick.

`OBJ.SendToNXT(HANDLE)` uses `HANDLE` to identifiy the connection to use for this command. This is optional. Otherwise the defaul handle (set using `COM_SetDefaultNXT`) will be used.

For a valid list of properties and how they affect the motors' behaviour, see the documentation for the class constructor `NXTMotor`.

Limitations

With `ActionAtTachoLimit = 'Coast'` and synchronous driving (two motors), the motors will stay synced after movement (even after `.WaitFor()` has finished). This is by design. To disable the synchonization, just use `.Stop`.

If you send a command to the NXT without waiting for the previous motor operation to have finished, the command will be dropped (the NXT indicates this with a high and low beep tone). Use the motor-objects method `.WaitFor` to make sure the motor is ready for new commands, or stop the motor(s) using `.Stop`.

Example

```
motor = NXTMotor('A', 'Power', 50, 'TachoLimit', 200);
motor.SendToNXT();
motor.WaitFor();
NXT_PlayTone(400,500);
```

See also

NXTMotor, ReadFromNXT, Stop, WaitFor, DirectMotorCommand,

Signature

- **Author:** Linus Atorf, Aulis Telle, Alexander Behrens (see AUTHORS)
- **Date:** 2009/07/12
- **Copyright:** 2007-2010, RWTH Aachen University

Published with wg_publish; V1.0

Stop

Stops or brakes specified motor(s)

Contents

- Syntax
- Description
- Example:
- See also
- Signature

Syntax

```
OBJ.Stop()

OBJ.Stop(BRAKEMODE)

OBJ.Stop(BRAKEMODE, HANDLE)
```

Description

`OBJ.Stop()` is the same as `OBJ.Stop('off')`.

`OBJ.Stop(BRAKEMODE)` stops the motor specified in `OBJ` with the brakemode specified in `BRAKEMODE`:

`BRAKEMODE` can take the following values:

`'nobrake'`, `'off'`, 0, `false`: The electrical power to the specified motor is simply disconnected, the so-called "COAST" mode. Motor will keep spinning until it comes to a soft stop.

`'brake'`, `'on'`, 1, `true`: This will actively halt the motor at the current position (until the next movement command); it's a "hard brake".

Use `HANDLE` to identify the connection to use for this command (optional).

Note:

To stop all motors at precisely the same time, please see the command `StopMotor`. It can be called with the syntax `StopMotor('all', 'off')` or `StopMotor('all', 'brake')`. When comparing this to the `obj.Stop` method, it acts more precise when wanting to stop multiple motors at the same time...

Using the active brake (e.g. `.Stop('brake')`) can be very power consuming, so watch your battery level when using this functionality for long periods of time.

Example:

```
motorC = NXTMotor('C', 'Power', -100, 'TachoLimit', 0);
motorC.SendToNXT();
pause(4);      % wait 4 seconds
motorC.Stop('brake');
```

See also

NXTMotor, **WaitFor**, **StopMotor**,

Signature

- **Author:** Aulis Telle, Linus Atorf (see AUTHORS)
- **Date:** 2009/07/20
- **Copyright:** 2007-2010, RWTH Aachen University

Published with wg_publish; V1.0

StopMotor

Stops / brakes specified motor. (Synchronisation will be lost after this)

Contents

- Syntax
- Description
- Limitations:
- Example
- See also
- Signature

Syntax

```
StopMotor(port, mode)

StopMotor(port, mode, handle)
```

Description

`StopMotor(port, mode)` stops the motor connected to the given `port`. The value `port` can be addressed by the symbolic constants `MOTOR_A`, `MOTOR_B`, `MOTOR_C` and `'all'` (all motor at at the same time) analog to the labeling on the NXT Brick. The argument `mode` can be equal to `'off'` (or `'nobrake'` or `false`) which cuts off the electrical power to the specific motor, so called "COAST" mode. The option `'brake'` (or `'on'` or `true`) will actively halt the motor at the current position (until the next command).

The last optional argument can be a valid NXT handle. If none is specified, the default handle will be used (call `COM_SetDefaultNXT` to set one).

Note:

The value `port` equal to `'all'` can be used to stopp all motors at the same time using only one single Bluetooth packet. After a `StopMotor` command the motor snychronization will be lost.

With `mode` equal to `'off'`, the motor will slowly stop spinning, but using `'brake'` applies real force to the motor to stand still at the current position, just like a real brake.

Using the active brake (e.g. `StopMotor(MOTOR_A, 'brake')`) can be very power consuming, so watch your battery level when using this functionality for long periods of time.

Limitations:

When working with a motor object that contains multiple motors (e.g. created by `NXTMotor('BC')`), stopping only one motor (in this case with e.g. `StopMotor(MOTOR_B, 'off')`) can lead to unexpected behavior. When working with synchronized motors, always stop those motors together. It's generally recommended to use the motor object's method `Stop` if possible.

Example

```
% regular stop
StopMotor(MOTOR_B, 'brake');
```

```
% imagine we have all motors moving at once:
m1 = NXTMotor('A',  'Power', 80);
m2 = NXTMotor('BC', 'Power', 50);
m1.SendToNXT();
m2.SendToNXT();

% a great way to stop all motors at once at the same time now:
StopMotor('all', 'off');

% the other possibility would not stop movement at precisely
% the same moment:
m1.Stop();
m2.Stop();
```

See also

NXTMotor, **Stop**, **NXT_SetOutputState**, **MOTOR_A**, **MOTOR_B**, **MOTOR_C**,

Signature

- **Author:** Linus Atorf, Alexander Behrens (see AUTHORS)
- **Date:** 2009/08/24
- **Copyright:** 2007-2010, RWTH Aachen University

Published with wg_publish; V1.0

SwitchLamp

Switches the LEGO lamp on or off (has to be connected to a motor port)

Contents

- Syntax
- Description
- Examples
- See also
- Signature

Syntax

```
SwitchLamp(port, mode)
```

```
SwitchLamp(port, mode, handle)
```

Description

`SwitchLamp(port, mode)` turns the LEGO lamp on or off. The given `port` number specifies the used motor port. The value `port` can be addressed by the symbolic constants `MOTOR_A` , `MOTOR_B` and `MOTOR_C` analog to the labeling on the NXT Brick. The value `mode` supports two modes `'on'` and `'off'` to turn the lamp on and off.

The last optional argument can be a valid NXT handle. If none is specified, the default handle will be used (call `COM_SetDefaultNXT` to set one).

This function simply sets power 100 to the specified motor port to turn on the lamp, or sets power 0 to turn it off. Note that dimming is not possible, even a power of just 1 will be enough to switch the lamp to full brightness (after a short while).

A `StopMotor` command with parameter `'off'` will also turn off the lamp, but it is recommended to use this function when working with lamps for better readability.

Examples

```
SwitchLamp(MOTOR_B, 'on');
```

```
SwitchLamp(MOTOR_A, 'off');
```

See also

NXTMotor, StopMotor, MOTOR_A, MOTOR_B, MOTOR_C,

Signature

- **Author:** Linus Atorf, Alexander Behrens (see AUTHORS)
- **Date:** 2007/10/15
- **Copyright:** 2007-2010, RWTH Aachen University

USGetSnapshotResults

Retrieves up to eight echos (distances) stored inside the US sensor

Contents

- Syntax
- Description
- Example
- See also
- Signature

Syntax

```
echos = USGetSnapshotResults(port)
```

```
echos = USGetSnapshotResults(port, handle)
```

Description

`echos = USGetSnapshotResults(port)` retrieves the echos originating from the ping that was sent out with `USMakeSnapshot(port)` for the ultrasonic sensor connected to `port`. In order for this to work, the sensor must have been opened in snapshot mode using `OpenUltrasonic(port, 'snapshot')`.

The last optional argument can be a valid NXT handle. If none is specified, the default handle will be used (call `COM_SetDefaultNXT` to set one).

The return-vector `echos` always contains exactly 8 readings for up to 8 echos that were recorded from one single signal. Depending on the structure and reflections, this can only be one valid echo, or a lot of interference. The values are distances, just as you'd expect from `GetUltrasonic`. The values are sorted in order of appearance, so they should also be sorted with increasing distances.

It is not known how exactly the measurements are to be interpreted!

Please make sure that after calling `USGetSnapshotResults`, there is a little time period for the sound waves to travel, to be reflected, and be recorded by the sensor. You can estimate this using the speed of sound and the maximum distance you expect to measure.

Note: When the US sensor is opened in snapshot mode, the function `GetUltrasonic` does not work correctly!

Example

```
port = SENSOR_4;
OpenUltrasonic(port, 'snapshot');
% send out the ping
USMakeSnapshot(port);
% wait some time for the sound to travel
pause(0.1); % 100ms is probably too much, calculate using c_sound ;-)
% retrieve all the echos in 1 step
echos = USGetSnapshotResults(port);
CloseSensor(SENSOR_4);
```

```
% You should also try the file Example_4_NextGenerationUltrasound.m
% from the demos-folder
```

See also

OpenUltrasonic, **GetUltrasonic**, **USMakeSnapshot**, **CloseSensor**,

Signature

- **Author:** Linus Atorf, Alexander Behrens (see AUTHORS)
- **Date:** 2008/01/08
- **Copyright:** 2007-2010, RWTH Aachen University

Published with wg_publish; V1.0

USMakeSnapshot

Causes the ultrasonic sensor to send one snapshot ("ping") and record the echos

Contents

- Syntax
- Description
- Example
- See also
- Signature

Syntax

```
USMakeSnapshot(port)

USMakeSnapshot(port, handle)
```

Description

Call `USMakeSnapshot(port)` to make the specific ultrasonic sensor connected to `port` send out one single signal (called "ping"). In contrast to the US sensor's continous mode (that is invoked by a normal `OpenUltrasonic` without the `'snapshot'` parameter), the sensor will only send out ultrasonic signals when this function is called. Up to 8 multiple echos will be recorded, and the according distances stored in the sensor's internal memory. Use `USGetSnapshotResults` to retrieve those 8 readings in one step!

The last optional argument can be a valid NXT handle. If none is specified, the default handle will be used (call `COM_SetDefaultNXT` to set one).

Note: When the US sensor is opened in snapshot mode, the function `GetUltrasonic` does not work correctly!

Example

```
port = SENSOR_4;
OpenUltrasonic(port, 'snapshot');
% send out the ping
USMakeSnapshot(port);
% wait some time for the sound to travel
pause(0.1); % 100ms is probably too much, calculate using c_sound ;-)
% retrieve all the echos in 1 step
echos = USGetSnapshotResults(port);
CloseSensor(SENSOR_4);
```

```
% You should also try the file Example_4_NextGenerationUltrasound.m
% from the demos-folder
```

See also

OpenUltrasonic, GetUltrasonic, USGetSnapshotResults, CloseSensor,

Signature

- **Author:** Linus Atorf, Alexander Behrens (see AUTHORS)
- **Date:** 2008/01/08
- **Copyright:** 2007-2010, RWTH Aachen University

Published with wg_publish; V1.0

WaitFor

Wait for motor(s) to stop (busy waiting)

Contents

- Syntax
- Description
- Examples:
- See also
- Signature

Syntax

```
OBJ.WaitFor TIMEDOUT = OBJ.WaitFor(TIMEOUT)

OBJ.WaitFor(HANDLE) TIMEDOUT = OBJ.WaitFor(TIMEOUT, HANDLE)
```

Description

`OBJ.WaitFor` waits for motor specified by `OBJ` to stop. We do this by reading the motor state from the NXT brick repeatedly until controlled movement is finished. If the motor is set to run infinitely, the method returns immediately and displays a warning.

`TIMEDOUT = OBJ.WaitFor(TIMEOUT)` does the same as described above but has an additional timeout `TIMEOUT` (given in seconds). After this time the function stops waiting and returns `true`. Otherwise it returns `false`. This functionality is useful to avoid that your robot (and your program) get stuck in case the motor should somehow get stalled (e.g.by driving against a wall).

Use `HANDLE` (optional) to identify the connection to use for this command.

Note:

If you specify `TIMEOUT` and the motor is not able to finish its current movement command in time (maybe because the motor is blocked?), waiting will be aborted. The motor is probably still busy in this case, so you have to make sure it is ready to accept commands before using it again (i.e. by calling `.Stop()`).

Examples:

```
% If a |SendToNXT| command is immediately followed by a |Stop|
% command without using |WaitFor| MATLAB does not wait to send
% the stop command until the motor has finished its rotation.
% Thus, the motor does not rotate at all, since the stop command
% reaches the NXT before the motor starts its rotation due to
% its mechanical inertia.
motorA = NXTMotor('A', 'Power', 60, 'TachoLimit', 1000);
motorA.SendToNXT();
motorA.Stop('off');
% motor A barely moved at all...

% To avoid this issue, WaitFor has to be used!
motorC = NXTMotor('C', 'Power', -20, 'TachoLimit', 120);
motorC.SendToNXT();
motorC.WaitFor();
data = motorC.ReadFromNXT();
```

```
% Instantiate motor A and run it
m = NXTMotor('A', 'Power', 50, 'TachoLimit', 1000);
m.SendToNXT();
% Wait for the motor, try waiting for max. 10 seconds
timedOut = WaitFor(m, 10);
if timedOut
    disp('Motor timed out, is it stalled?')
    m.Stop('off'); % this needed to "unlock" the motor
end%if
% now we can send new motor commands again...
```

See also

NXTMotor, **ReadFromNXT**, **SendToNXT**, **Stop**, **StopMotor**,

Signature

- **Author:** Aulis Telle, Linus Atorf (see AUTHORS)
- **Date:** 2009/07/20
- **Copyright:** 2007-2010, RWTH Aachen University

Published with wg_publish; V1.0

checkStatusByte

Interpretes the status byte of a return package, returns error message

Contents

- Syntax
- Description
- Example
- See also
- Signature

Syntax

```
[flag err_message] = checkStatusByte(response)
```

Description

`[flag err_message] = checkStatusByte(response)` interpretes the `response` byte (not hexadecimal). The return value `flag` indicates wether an error has occured (`true` = error, `false` = success). The string value `err_message` contains the error message (or '' in case of success).

Example

```
[status SleepTimeLimit] = NXT_SendKeepAlive('reply');
[err errmsg] = checkStatusByte(status);
```

See also

COM_CollectPacket, NXT_LSGetStatus,

Signature

- **Author:** Linus Atorf (see AUTHORS)
- **Date:** 2007/10/15
- **Copyright:** 2007-2010, RWTH Aachen University

Published with wg_publish; V1.0

readFromIniFile

Reads parameters from a configuration file (usually *.ini)

Contents

- Syntax
- Description
- Examples:
- Signature

Syntax

```
ret = readFromIniFile(AppName, KeyName, filename)
```

Description

ret=readFromIniFile(AppName, KeyName, filename)

This function works like GetPrivateProfileString() from the Windows-API that is well known for reading ini-file data. The parameters are:

AppName = Section name of the type [xxxx] KeyName = Key name separated by an equal to sign, of the type abc = 123 filename = ini file name to be used

ret = string (!) value of the key found, empty ('') if key was not found. Since it's a string, you have to convert to integers / floats on your own.

Note that AppName and KeyName are NOT case sensitive, and that whitespace between '=' and the value will be ignored (removed).

If the key or the AppName is not found, or if the inifile does not exist or could not be opened, '' will be returned (this will also be returned when the key is empty).

Examples:

```
%a sample ini file (sample.ini) may have entries:
%
%   [XYZ]
%   abc=123
%   def=7
%   ; lines starting with a ; are comments
%   [ZZZ]
%   abc=890
```

```
readFromIniFile('XYZ','abc', 'sample.ini') % will return '123'
```

```
readFromIniFile('ZZZ','abc', 'sample.ini') % will return '890'
```

```
readFromIniFile('XYZ','def', 'sample.ini') % will return '7'
```

Signature

- **Author:** Linus Atorf (see AUTHORS)
- **Date:** 2008/01/08
- **Copyright:** 2007-2010, RWTH Aachen University

Published with wg_publish; V1.0

textOut

Wrapper for fprintf() which can optionally write screen output to a logfile

Contents

- Syntax
- Description
- Examples
- See also
- Signature

Syntax

```
textOut(strMsg)

textOut(strMsg, 'screenonly')

textOut(strMsg, 'logonly')
```

Description

`textOut(strMsg)` will write to screen (command window) AND logfile, if global logging is enabled.

`textOut(strMsg, 'screenonly')` writes to screen (command window) only.

`textOut(strMsg, 'logonly')` writes to logfile only (global logging must be enabled). If logging is disabled or somehow failes, the message will not be logged or displayed at all...

Note:

To enable logging (to file), set the global var `EnableLogging` to `true` and set `Logfilename` to a valid filename. This function distinguishes between Windows and Linux systems to use proper linebreaks. The global variable `DisableScreenOut` is an on/off switch for the complete textOut()-messages that would appear on the command window. Default setting is `false`, i.e. there will be no output.

Recommended usage is together with sprintf(), in order to add linebreaks for example.

Examples

```
textOut('This is a message\n');
```

```
x = 'world';
y = 2007;
textOut(sprintf('Hello %s, it is the year %d!\n', x, y));
%Results in:  >> Hello world, it is the year 2007!
```

```
global EnableLogging
global Logfilename
EnableLogging = true;
Logfilename = 'logfile.txt';
textOut(sprintf('Whatever I say here will be logged to the file as well.\n'));
```

```
textOut(sprintf('Only appears in the command window\n'), 'screenonly');
```

```
textOut(sprintf('Only appears in the logfile, if logging enabled\n'), 'logonly');
```

```
global DisableScreenOut
DisableScreenOut = true;
textOut(sprintf(['If logging is enabled, this goes to the logfile, ', ...
                 'if not, this goes nowhere\n']));
```

See also

DebugMode, **isdebug (private)**,

Signature

- **Author:** Linus Atorf (see AUTHORS)
- **Date:** 2007/10/15
- **Copyright:** 2007-2010, RWTH Aachen University

Published with wg_publish; V1.0

Command Layer Structure

The functions of the RWTH - Mindstorms NXT Toolbox can be categorized into a multiple layer structure. On the lowest layer **Low Level and Helper Functions** are available, which mostly convert parameter modes to bytes words, determined by the LEGO direct commands documentation. The second layer includes **Direct NXT Commands** which are mapped from the LEGO direct command documentation without any limitations and can be identified by the `NXT_*` prefix. Also Bluetooth packet related functions can be found in this layer. Layer 3 provides **High Level Functions** for controlling the NXT motors, sensors and the Bluetooth connection. These functions are basically using the Direct NXT Commands of layer 2 to make the motor and sensor controlling more convenient and easily readable for the user. The top layer provides **High Level Regulation** functions for precise motor regulation and various utilities.

Layer	Description	Output / Motors	Input / Sensors	General	Bluetooth / USB
4	**High Level Regulation / Utilities**	**NXTMotor** .ReadFromNXT .SendToNXT .Stop .WaitFor .ResetPosition **NXC_MotorControl**		OptimizeToolboxPerformance **GUI_WatchMotorState** **GUI_WatchSensor**	**COM_MakeBTConfigFile**
		DirectMotorCommand **StopMotor** **SwitchLamp** NXC_ResetErrorCorrection	**OpenLight** **GetLight** **OpenSound** **GetSound** **OpenSwitch** **GetSwitch** **OpenUltrasonic** **GetUltrasonic** **USMakeSnapshot** **USGetSnapshotResults** **OpenAccelerator** **GetAccelerator** **OpenColor**	**readFromIniFile** MAP_GetCommModule MAP_GetInputModule MAP_GetOutputModule MAP_GetSoundModule MAP_GetUIModule MAP_SetOutputModule NXC_GetSensorMotorData	**COM_OpenNXT** **COM_OpenNXTEx** **COM_CloseNXT** COM_ReadI2C

3	**High Level Functions**		**CalibrateColor** **GetColor** **OpenNXT2Color** **GetNXT2Color** **OpenCompass** **CalibrateCompass** **GetCompass** **OpenGyro** **CalibrateGyro** **GetGyro** **OpenEOPD** **CalibrateEOPD** **GetEOPD** **OpenInfrared** **GetInfrared** **OpenRFID** **GetRFID** **CloseSensor**		
2	**Direct NXT Commands**	**NXT_SetOutputState** **NXT_GetOutputState** **NXT_ResetMotorPosition**	**NXT_SetInputMode** **NXT_GetInputValues** NXT_ResetInputScaledValue NXT_LSRead NXT_LSWrite NXT_LSGetStatus	**NXT_PlayTone** NXT_PlaySoundFile NXT_StopSoundPlayback NXT_StartProgram NXT_GetCurrentProgramName NXT_StopProgram NXT_SendKeepAlive NXT_GetBatteryLevel NXT_GetFirmwareVersion NXT_SetBrickName NXT_ReadIOMap NXT_WriteIOMap NXT_MessageWrite NXT_MessageRead	COM_CreatePacket COM_SendPacket COM_CollectPacket **COM_SetDefaultNXT** COM_GetDefaultNXT
		MOTOR_A **MOTOR_B**	**SENSOR_1** **SENSOR_2**	**DebugMode** *isdebug*	checkStatusByte

1	**Low Level Functions: Helper, Conversion and Lookup Functions**	**MOTOR_C** *SetMotor* (o) *SetPower* (o) *SetAngleLimit* (o) *SetRampMode* (o) *SetTurnRatio* (o) *SpeedRegulation* (o) *SyncToMotor* (o) *GetMotor* (o) *SetMemoryCount* (o) *GetMemoryCount* (o) *byte2outputmode* *byte2regmode* *byte2runstate* *outputmode2byte* *regmode2byte* *runstate2byte* *initializeGlobalMotorStateVar* (o)	**SENSOR_3** **SENSOR_4** *byte2sensortype* *byte2sensormode* *sensortype2byte* *sensormode2byte* *waitUntilI2CReady*	textOut *dec2wordbytes* *name2commandbytes* *commandbyte2name* *wordbytes2dec*	*createHandleStruct* *checkHandleStruct* *getLibusbErrorString* *getVISAErrorString* *getReplyLengthFromCmdByte* *fantom_proto* *libusb_proto*

legend: NXT_* = NXT Direct commands without any limitations (mapped to the LEGO direct command documentation)
COM_* = Functions related to the NXT communication
MAP_* = Functions related to the NXT module maps
NXC_* = Functions communicating with the embedded NXC program MotorControl
bold = Main funcions or main group functions
italic = private functions
(o) = obsolete / deprecated functions (might be removed in a future release)