Bohrvorrichtung Documentation

Release 0.0.1

Simon Heider

Nov 18, 2016

Contents:

CHAPTER

ONE

ÜBERBLICK

Das nachfolende Dokument soll die Funktionsweise, sowie den Quellcode der Felgenbohrmaschine dokumentieren. Die Bohrmaschine besteht im wesentlichen aus zwei Schrittmotoren. Einen zentralen Rotor, welcher für die Rotation und Positionierung der Felge zuständig ist und einer Lineareinheit, welche den Dremel positioniert und die Bohrbewegung ausführt.

1.1 Ansteuerung der Motoren

In den Motoren sind mehrere Datensätze für zugehörige Motoroperationen gespeichert. Die wichtigsten Parameter eines Datensatzes sind:

- Schrittversatz: Die Positionsveränderung in Schritten
- Geschwindigkeit: Die Geschwindigkeit der Positionsveränderung in Hz (Schritte pro Sekunde)
- Bewegungsmodus: Inkremental (0) oder Absolut (1). Bei einer inkrementalen Bewegung werden die Schritte im Schrittversatz zur aktuellen Position hinzuaddiert. Bei einer absoluten Bewegung positioniert sich der Motor gemessen am Ausgangspunkt.

Soll nun eine bestimmte Operation ausgeführt werden, so muss am Inputregister die entsprechende Operation ausgewählt werden und das Startflag gesetzt werden.

Zusätzlich zu den gespeicherten Operationen kann eine ReturnToHome Operation durchgeführt werden. Hierbei fährt der Motor auf die Nullposition zurück. Die ReturnToHome Operation wird gestartet, indem das entsprechnede Bit am Inputregister gesetzt wird.

Eine genauere Erläuterung zur Ansteuerung finded sich im Kapitel *Modbus/RS-485*, sowie im OrientalMotor Datenblatt Kapitel 5 und in der Softwaredokumentation.

1.1.1 Rotor

Der zentrale Rotor besitzt nur einen Datensatz, welcher unter dem Index 0 gespeichert ist. Die zugehörige Operation rotiert die Felge um einen Bohrversatz.

1.1.2 Lineareinheit

In der Lineareinheit sind folgende Datensätze hinterlegt:

- Index 0: Verfahren zum Bohrausgangspunkt (Pos. 1 mit Geschw. 1)
- Index 1: Verfahren zum Bohrmittelpunkt (Pos. 2 mit Geschw. 2)
- Index 3: Verfahren zum Bohrendpunkt (Pos. 3 mit Geschw. 3)

- Index 4: Rückbewegung zum Bohrausgangspunkt (Pos. 1 mit Geschw. 4)
- Index 5: Verfahren zum Linearnullpunkt UNKALIBRIERT

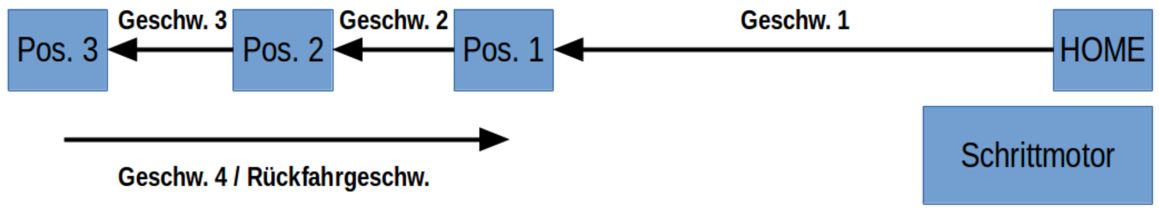

Fig. 1.1: Positionen und Geschwindigkeiten der Lineareinheit

1.2 Motorkonfiguration

Für eine funktionsfähige und fehlerfreie Kommunikation zwischen Steuerung und Schrittmotortreiber müssen die Treiber richtig konfiguriert sein. Folgende Einstellungen müssen überwacht werden:

- ID: Adresse des Motors im Modbusprotokoll
- Transmission rate (SW2): Serielle Übertragungsgeschwindigkeit
- Function settings switch (SW4): Protokollkonfiguration
- Terminierungswiderstände (TERM): Konfiguration der Terminierungswiderstände

1.2.1 Rotor

- ID: 5
- SW2: 1 (19200 BAUD)
- SW4-No.1: OFF
- SW4-No.2: ON
- TERM-No.1: ON
- TERM-No.2: OFF

1.2.2 Lineareinheit

- ID: 6
- SW2: 1 (19200 BAUD)
- SW4-No.1: OFF
- SW4-No.2: ON
- TERM-No.1: ON

- TERM-No.2: ON

1.3 Modbus/RS-485

Die in *Ansteuerung der Motoren* beschriebenen Datensätze lassen sich mittels Motbus-RTU/RS-485 manipulieren. Modbus ist ein serielles Übertragungsprotokoll auf dem Master-Slave-Prinzip. Ein Master schickt Anforderungen an einen definierten Slave. Hat der Slave die Anfrage erhalten und bearbeitet schickt dieser eine Antwort an den Master. Die Steuerung der Felgenbohrmaschine benutzt folgende Anfragen:

- writeRegister: Schreibt einen Wert in ein Register des Motortreibers
- readRegister: Liest einen Wert aus einem Register des Motortreibers

In den verschiedenen Treiberregistern werden Prozessparameter, wie z.B. Verfahrschritte, Verfahrgeschwindigkeit oder Beschleunigung des Motors gespeichert. Über das Inputregister (0x125) lässt sich der Motor starten und stoppen. Über das Outputregister (0x127) können mittels readRegister Statusinformationen über den Motor abgefragen werden. Eine genaue Dokumentation der Register und deren Inhalte, sowie zum Modbusprotokoll findet sich im OrientalMotor Datenblatt Kapitel 5.

1.4 Aufbau der Mastersteuerung/Software

Als Master der Modbuskommunikation kommt ein Rasperry Pi (RP) mit nachfolgend beschriebener Software zum Einsatz. Über einen USB zu RS-485 Konverter wird eine Verbindung zwichen RP und Schrittmotoren hergestellt. Der Konverter muss wie folgt konfiguriert sein:

- SW1: OFF
- SW2: OFF
- SW3: OFF
- SW4: OFF

Auf dem RP läuft das Hauptprogramm der Bohrvorrichtung (`bohrvorrichtung`), welche auf einkommende Kommandos höhrt. Sobald ein Kommando empfangen wurde, wird dieses ausgeführt.

Des Weiteren läuft ein Listenerprogramm, welches darauf wartet, dass der Druckknopf zum Starten gedrückt wird (`button`). Wird der Knopf gedrückt, so sendet das Programm ein Startsignal an das Hauptprogramm.

Außerdem kann manuell die grafische Operfläche gestartet werden (`gui`). Die grafische Oberfläche stellt ebenfalls eine Verbindung zum Hauptprogramm her und sendet je nach Benutzereingabe verschiedene Signale an das Hauptprogramm.

1.4.1 Aufbau des Hauptprogramms

Wie bereits in *Modbus/RS-485* erwähnt, werden die Schrittmotoren über das Modbusprotokoll angesteuert. `minimalmodbus` stellt grundlegende Funktionalitäten zur Modbuskommunikation zur Verfügung. Beispielsweise writeRegister (zum Schreiben von Prozessparametern) oder readRegister (zum Lesen von Prozessparametern). `stepperMotor` erbt von `minimalmodbus` und stellt einen Schrittmotor dar. Mithilfe von `stepperMotor` kann ein Schrittmotor angesteuert, also bewegt und konfiguriert, werden. Das Hauptprogramm (`bohrvorrichtung`) konstruiert sich nun zwei Instanzen von `stepperMotor`, welche jeweils den Rotor beziehungsweise die Lineareinheit darstellen. Beim Initialisieren werden alle benötigten Prozessparameter aus der Datei "processData.json" geladen. Die Datei entspricht dem JSON Standard. Der gültige Aufbau wird in `bohrvorrichtung` dokumentiert. Nach dem Initialisieren wird die Hauptschleife gestartet, welche auf einkommende Kommandos höhrt (definiert in `commands`).

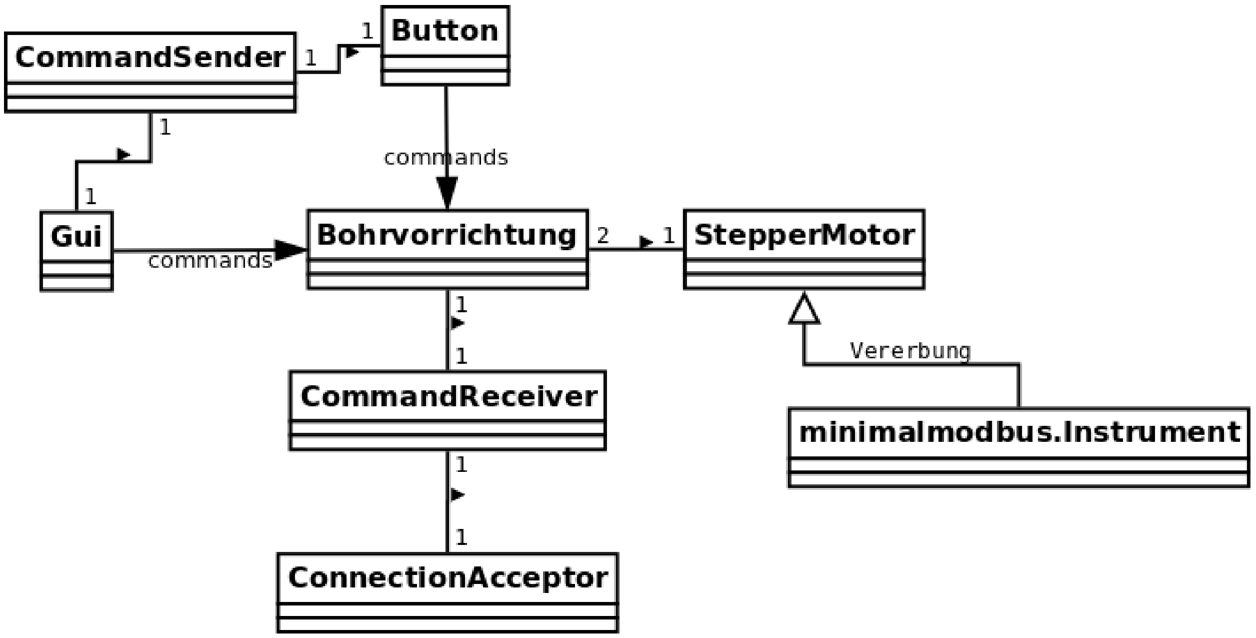

Fig. 1.2: UML Diagramm der Steuerungssoftware

1.4.2 Aufbau der Interprozesskommunikation

Die Kommunikation zwischen `gui`, `button` und `bohrvorrichtung` wird von `communicationUtilities` gesteuert. `bohrvorrichtung` besitzt eine Instanz der Klasse `communicationUtilities.CommandReceiver` welche auf neue Verbindungen höhrt und Kommandos empfängt. Das Hauptprogramm kann dann empfangene Befehle aus der Warteschlange von `communicationUtilities.CommandReceiver` nehmen und nach deren Ausführung die entsprechende Antwort in die Ausgangswarteschlange von `communicationUtilities.CommandReceiver` schreiben.

`gui` und `button` besitzen jeweils eine Instanz der Klasse `communicationUtilities.CommandSender`, welche zuerst eine Verbindung zu `communicationUtilities.CommandReceiver` aufbaut und danach Befehle an die Gegenseite verschicken kann. Eine Sender darf erst ein neues Kommando verschicken, wenn ein Rückmeldung vom alten Kommando erhalten wurde. Ein Interrupt zum Stop der Maschine darf jedoch jederzeit gesendet werden.

CHAPTER

TWO

BOHRVORRICHTUNG QUELLCODEDOKUMENTATION

2.1 bohrvorrichtung module

class `bohrvorrichtung.`**`Bohrvorrichtung`**

Bases: `object`

The actual driller. Starts listening on commands when *`start()`* is called!

Attributes:

- rotor: The rotor stepper motor
- linear: The linear stepper motor
- isInterrupted: Flag if driller is interrupted or not
- mainLoopWaitTime: Timeout for mainloop
- cr: Instance of CommandReceiver; Handles all incoming command requests

`__init__``()`

Constructor. Initializes everythin.

`driveLinearTo``(`*position*`)`

Drives linear to given position.

`handleCommand``(`*command*`)`

Parses a incomming command and performs an action.

Returns: SUCCESS, FAIL or INVALID_REQUEST

`handleInterrupt``()`

Interrupts the driller. All motors are STOPPED.

`loadProcessData``()`

Loads process data from json file. The json file must define:

Attributes:

- holeNumber: The ammount of holes
- rotorSteps: The steps for each rotor move
- rotorOperationSpeed
- rotorOperationMode
- x1: The position where drilling starts
- x2: The middleposition of drilling
- x3: The position where drilling ends

- x4: The position where drilling starts
- x5: Nullposition
- v1: Speed to x1
- v2: Speed to x2
- v3: Speed to x3
- v4: Speed to x4
- v5: Speed to x5
- mode1
- mode2
- mode3
- mode4
- mode5

`processDataToDevice``()`
Loads and writes process data.

`sayHello``()`
Process initial movement.

`start``()`
Starts mainloop and listens to incoming commands. For valid commands see *`commands`*

`startDrilling``()`
Perform drilling process.

`writeProcessData``()`
Writes process data to stepper motor registers.

2.2 button module

class `button.`**`Button`**`(`*buttonPin=18*`)`
Bases: `object`

Button which is attached to rasperry pi.

Attributes:

- buttonPin: GPIO pin where the button is connected to
- allowedToSend: indicates if button is allowed to send a command or not; Button is not allowed to change this value by itself
- cs: An instance of CommandSender, which sends starting commands to the driller

`__init__``(`*buttonPin=18*`)`
Constructor; sets attributes.

Args: see attributes

Returns: None

Raises: None

onResponse(*response*)
Printing the command response.

start()
Listens if the button is pressed and sends a starting command to the driller, if its allowed to.

2.3 commands module

This module defines all command request a command sender (such as gui or button) can send to the driller and their responses.

commands.**INTERRUPT = 'interrupt'**
The driller is interrupted and stops all movement.

commands.**REQUEST_DRIVEX1 = 'driveToX1'**
Linear is driving to startposition of drilling.

commands.**REQUEST_DRIVEX2 = 'driveToX2'**
Linear is driving to middleposition of drilling.

commands.**REQUEST_DRIVEX3 = 'driveToX3'**
Linear is driving to endposition of drilling.

commands.**REQUEST_RELOADDATA = 'reloadData'**
The Processdata is reloaded from the processData file and is written to the motors.

commands.**REQUEST_STARTDRILLING = 'startDrilling'**
Starts the drilling process.

commands.**RESPONSE_FAIL = 'fail'**
When the operation was unsuccessfull.

commands.**RESPONSE_INVALID_REQUEST = 'invalidRequest'**
When a invalid request was send.

commands.**RESPONSE_SUCCESS = 'success'**
When the operation was successfull.

2.4 communicationUtilities module

class communicationUtilities.**CommandReceiver**(*master*, *host='‘*, *port=54321*, *timeout=0.2*)
Bases: threading.Thread

Receives command requests and sends command responses.

Attributes:

- master: A class that handles the commands; must have a boolean attribute named "isInterrupted"
- connections: A list where all connections are stored
- commandRequests: A queue where all received (commandRequests, con) are stored
- commandResponses: A queue where all (commandResponses, con) are stored, which have to be send
- timeout: Blocking time for reading sockets
- ca: An instance of ConnectionAcceptor which accepts new connections and writes them to connections

`__init__`(*master*, *host=''*, *port=54321*, *timeout=0.2*)

Constructor; sets attributes and starts ConnectionAcceptor

Args: see Attributes

Returns: None

Raises: None

`receiveCommandRequests`()

Reads all incoming command requests and writes them to commandRequests. Interrupts the master, if necessary.

Args: None

Returns: None

Raises: None

`run`()

Starts main receiving and sending loop in a thread.

`sendCommandResponses`()

Sends all command responses stored in commandResponses.

Args: None

Returns: None

Raises: None

class `communicationUtilities.`**`CommandSender`**(*master*, *host='localhost'*, *port=54321*, *timeout=0.2*)

Bases: `threading.Thread`

Class to send command requests and wait for command responses.

Attributes:

- master: A class which wants to send command requests; must have a boolean attributes named "allowedToSend" and and implement a method onResponse(response)
- sendInterrupt: If true CommandSender sends an interrupt command to CommandReceiver
- timeout: Blocking time for reading socket
- s: socket, which is connected to a commandReceiver
- commandRequestToSend: The command which will be send in the next iteration of the main loop

`__init__`(*master*, *host='localhost'*, *port=54321*, *timeout=0.2*)

Constructor; sets attributes and connects the socket to a CommandReceiver.

Args:

- host: host to connect to
- port: port to connect to
- rest: see attributes

Returns: None

Raises: None

`receiveResponse`()

Waits until a command response has arrived. Sends an interrupt command if necessary.

Args: None

Returns: None

Raises: None

run()

sendRequest()
Sends the commandRequestToSend request to a CommandReceiver. Waits for the response in a thread.

Args: None

Returns: None

Raises: None

class `communicationUtilities.`**ConnectionAcceptor**(*master*, *host*, *port*)
Bases: `threading.Thread`

Class for accepting new socet connections in a thread.

Attributes:

- master: Class which handles the connections; must have a list as attribute named "connections"
- host: host ipadress
- port: port to listen

__init__(*master*, *host*, *port*)
Constructor; sets the attributes. Args:

see Attributes

Returns: None

Raises: None

run()
Start main listeningloop and writes new connections to connectionWorker.connections.

Args: None

Returns: None

Raises: None

2.5 gui module

class `gui.`**Gui**
Bases: `PyQt4.QtGui.QDialog`

The gui to configure the driller.

Attributes:

- receivedResponse: The signal which is emmited if the gui has received an answer
- ui: The interface
- cs: The sender which sends commands to the driller
- allowedToSend: A flag which indicates, if the gui is allowed to send commands
- allProcessData: A list of dictionarys which stores all process data. See *loadAllProcessData()*

- allProcessDataFileName: The json file where all process data are stored

`__init__``()`

`closeEvent``()`

`disableWidgets``()`
Disables all widgets.

`enableWidgets``()`
Enables all widgets.

`getDataFromEdits``()`
Reads from input edits.

Args: None

Returns: A dict with process data information. See *`loadAllProcessData()`*

Raises: ValueError

`getProcessDataByName``(`*name*`)`

`loadAllProcessData``()`
Loads all process data from the json file and stores them into allProcessData. One process data must define the following:

Attributes:

- holeNumber: The ammount of holes
- rotorSteps: The steps for each rotor move
- rotorOperationSpeed
- rotorOperationMode
- x1: The position where drilling starts
- x2: The middleposition of drilling
- x3: The position where drilling ends
- x4: The position where drilling starts
- x5: Nullposition
- v1: Speed to x1
- v2: Speed to x2
- v3: Speed to x3
- v4: Speed to x4
- v5: Speed to x5
- mode1
- mode2
- mode3
- mode4
- mode5

`onComboBoxChanged``()`
Updates gui after combo box index change.

onDeleteButtonClicked()
Deletes current process data and updates allProcess data file.

onDriveToX1ButtonClicked()

onDriveToX2ButtonClicked()

onDriveToX3ButtonClicked()

onReceiveResponseEvent(*response*)
Gets called when a response was received. Displays respone information and enables widges.

onResponse(*response*)

onSaveButtonClicked()
Saves current process data to allProcessData and writes it to file.

onSaveDataAndLoadButtonClicked()
Saves process data and sends reload command.

onStartButtonClicked()

onStopButtonClicked()

receivedResponse

refreshDataEdits(*processData*)
Refreshes input edits with given values.

Args:

- processData: A dict with process data. See `loadAllProcessData()`

Returns: None

Raises: None

2.6 minimalmodbus module

MinimalModbus: A Python driver for the Modbus RTU and Modbus ASCII protocols via serial port (via RS485 or RS232).

minimalmodbus.**BAUDRATE = 19200**
Default value for the baudrate in Baud (int).

minimalmodbus.**BYTESIZE = 8**
Default value for the bytesize (int).

minimalmodbus.**CLOSE_PORT_AFTER_EACH_CALL = False**
Default value for port closure setting.

class minimalmodbus.**Instrument**(*port*, *slaveaddress*, *mode='rtu'*)
Bases: `object`

Instrument class for talking to instruments (slaves) via the Modbus RTU or ASCII protocols (via RS485 or RS232).

Args:

- port (str): The serial port name, for example `/dev/ttyUSB0` (Linux), `/dev/tty.usbserial` (OS X) or `COM4` (Windows).
- slaveaddress (int): Slave address in the range 1 to 247 (use decimal numbers, not hex).

- mode (str): Mode selection. Can be MODE_RTU or MODE_ASCII.

`__init__` (*port*, *slaveaddress*, *mode='rtu'*)

`address` **= None**
Slave address (int). Most often set by the constructor (see the class documentation).

`close_port_after_each_call` **= None**
If this is `True`, the serial port will be closed after each call. Defaults to *`CLOSE_PORT_AFTER_EACH_CALL`*. To change it, set the value `minimalmodbus.CLOSE_PORT_AFTER_EACH_CALL=True` .

`debug` **= None**
Set this to `True` to print the communication details. Defaults to `False`.

`handle_local_echo` **= None**
Set to to `True` if your RS-485 adaptor has local echo enabled. Then the transmitted message will immeadiately appear at the receive line of the RS-485 adaptor. MinimalModbus will then read and discard this data, before reading the data from the slave. Defaults to `False`.

New in version 0.7.

`mode` **= None**
Slave mode (str), can be MODE_RTU or MODE_ASCII. Most often set by the constructor (see the class documentation).

New in version 0.6.

`precalculate_read_size` **= None**
If this is `False`, the serial port reads until timeout instead of just reading a specific number of bytes. Defaults to `True`.

New in version 0.5.

`read_bit` (*registeraddress*, *functioncode=2*)
Read one bit from the slave.

Args:

- registeraddress (int): The slave register address (use decimal numbers, not hex).
- functioncode (int): Modbus function code. Can be 1 or 2.

Returns: The bit value 0 or 1 (int).

Raises: ValueError, TypeError, IOError

`read_float` (*registeraddress*, *functioncode=3*, *numberOfRegisters=2*)
Read a floating point number from the slave.

Floats are stored in two or more consecutive 16-bit registers in the slave. The encoding is according to the standard IEEE 754.

There are differences in the byte order used by different manufacturers. A floating point value of 1.0 is encoded (in single precision) as 3f800000 (hex). In this implementation the data will be sent as `'\x3f\x80'` and `'\x00\x00'` to two consecutetive registers . Make sure to test that it makes sense for your instrument. It is pretty straight-forward to change this code if some other byte order is required by anyone (see support section).

Args:

- registeraddress (int): The slave register start address (use decimal numbers, not hex).
- functioncode (int): Modbus function code. Can be 3 or 4.

- numberOfRegisters (int): The number of registers allocated for the float. Can be 2 or 4.

Type of floating point number in slave	Size	Registers	Range
Single precision (binary32)	32 bits (4 bytes)	2 registers	1.4E-45 to 3.4E38
Double precision (binary64)	64 bits (8 bytes)	4 registers	5E-324 to 1.8E308

Returns: The numerical value (float).

Raises: ValueError, TypeError, IOError

`read_long` (*registeraddress*, *functioncode=3*, *signed=False*)
Read a long integer (32 bits) from the slave.

Long integers (32 bits = 4 bytes) are stored in two consecutive 16-bit registers in the slave.

Args:

- registeraddress (int): The slave register start address (use decimal numbers, not hex).
- functioncode (int): Modbus function code. Can be 3 or 4.
- signed (bool): Whether the data should be interpreted as unsigned or signed.

`signed`	Data type in slave	Alternative name	Range
`False`	Unsigned INT32	Unsigned long	0 to 4294967295
`True`	INT32	Long	-2147483648 to 2147483647

Returns: The numerical value (int).

Raises: ValueError, TypeError, IOError

`read_register` (*registeraddress*, *numberOfDecimals=0*, *functioncode=3*, *signed=False*)
Read an integer from one 16-bit register in the slave, possibly scaling it.

The slave register can hold integer values in the range 0 to 65535 ("Unsigned INT16").

Args:

- registeraddress (int): The slave register address (use decimal numbers, not hex).
- numberOfDecimals (int): The number of decimals for content conversion.
- functioncode (int): Modbus function code. Can be 3 or 4.
- signed (bool): Whether the data should be interpreted as unsigned or signed.

If a value of 77.0 is stored internally in the slave register as 770, then use `numberOfDecimals=1` which will divide the received data by 10 before returning the value.

Similarly `numberOfDecimals=2` will divide the received data by 100 before returning the value.

Some manufacturers allow negative values for some registers. Instead of an allowed integer range 0 to 65535, a range -32768 to 32767 is allowed. This is implemented as any received value in the upper range (32768 to 65535) is interpreted as negative value (in the range -32768 to -1).

Use the parameter `signed=True` if reading from a register that can hold negative values. Then upper range data will be automatically converted into negative return values (two's complement).

`signed`	Data type in slave	Alternative name	Range
`False`	Unsigned INT16	Unsigned short	0 to 65535
`True`	INT16	Short	-32768 to 32767

Returns: The register data in numerical value (int or float).

Raises: ValueError, TypeError, IOError

read_registers(*registeraddress*, *numberOfRegisters*, *functioncode=3*)
Read integers from 16-bit registers in the slave.

The slave registers can hold integer values in the range 0 to 65535 (“Unsigned INT16”).

Args:

- registeraddress (int): The slave register start address (use decimal numbers, not hex).
- numberOfRegisters (int): The number of registers to read.
- functioncode (int): Modbus function code. Can be 3 or 4.

Any scaling of the register data, or converting it to negative number (two’s complement) must be done manually.

Returns: The register data (a list of int).

Raises: ValueError, TypeError, IOError

read_string(*registeraddress*, *numberOfRegisters=16*, *functioncode=3*)
Read a string from the slave.

Each 16-bit register in the slave are interpreted as two characters (1 byte = 8 bits). For example 16 consecutive registers can hold 32 characters (32 bytes).

Args:

- registeraddress (int): The slave register start address (use decimal numbers, not hex).
- numberOfRegisters (int): The number of registers allocated for the string.
- functioncode (int): Modbus function code. Can be 3 or 4.

Returns: The string (str).

Raises: ValueError, TypeError, IOError

write_bit(*registeraddress*, *value*, *functioncode=5*)
Write one bit to the slave.

Args:

- registeraddress (int): The slave register address (use decimal numbers, not hex).
- value (int): 0 or 1
- functioncode (int): Modbus function code. Can be 5 or 15.

Returns: None

Raises: ValueError, TypeError, IOError

write_float(*registeraddress*, *value*, *numberOfRegisters=2*)
Write a floating point number to the slave.

Floats are stored in two or more consecutive 16-bit registers in the slave.

Uses Modbus function code 16.

For discussion on precision, number of registers and on byte order, see `read_float()`.

Args:

- registeraddress (int): The slave register start address (use decimal numbers, not hex).
- value (float or int): The value to store in the slave

- numberOfRegisters (int): The number of registers allocated for the float. Can be 2 or 4.

Returns: None

Raises: ValueError, TypeError, IOError

`write_long`(*registeraddress*, *value*, *signed=False*)

Write a long integer (32 bits) to the slave.

Long integers (32 bits = 4 bytes) are stored in two consecutive 16-bit registers in the slave.

Uses Modbus function code 16.

For discussion on number of bits, number of registers, the range and on alternative names, see `read_long()`.

Args:

- registeraddress (int): The slave register start address (use decimal numbers, not hex).
- value (int or long): The value to store in the slave.
- signed (bool): Whether the data should be interpreted as unsigned or signed.

Returns: None

Raises: ValueError, TypeError, IOError

`write_register`(*registeraddress*, *value*, *numberOfDecimals=0*, *functioncode=16*, *signed=False*)

Write an integer to one 16-bit register in the slave, possibly scaling it.

The slave register can hold integer values in the range 0 to 65535 (“Unsigned INT16”).

Args:

- registeraddress (int): The slave register address (use decimal numbers, not hex).
- value (int or float): The value to store in the slave register (might be scaled before sending).
- numberOfDecimals (int): The number of decimals for content conversion.
- functioncode (int): Modbus function code. Can be 6 or 16.
- signed (bool): Whether the data should be interpreted as unsigned or signed.

To store for example `value=77.0`, use `numberOfDecimals=1` if the slave register will hold it as 770 internally. This will multiply `value` by 10 before sending it to the slave register.

Similarly `numberOfDecimals=2` will multiply `value` by 100 before sending it to the slave register.

For discussion on negative values, the range and on alternative names, see `read_register()`.

Use the parameter `signed=True` if writing to a register that can hold negative values. Then negative input will be automatically converted into upper range data (two’s complement).

Returns: None

Raises: ValueError, TypeError, IOError

`write_registers`(*registeraddress*, *values*)

Write integers to 16-bit registers in the slave.

The slave register can hold integer values in the range 0 to 65535 (“Unsigned INT16”).

Uses Modbus function code 16.

The number of registers that will be written is defined by the length of the `values` list.

Args:

- registeraddress (int): The slave register start address (use decimal numbers, not hex).
- values (list of int): The values to store in the slave registers.

Any scaling of the register data, or converting it to negative number (two's complement) must be done manually.

Returns: None

Raises: ValueError, TypeError, IOError

`write_string`(*registeraddress*, *textstring*, *numberOfRegisters=16*)
Write a string to the slave.

Each 16-bit register in the slave are interpreted as two characters (1 byte = 8 bits). For example 16 consecutive registers can hold 32 characters (32 bytes).

Uses Modbus function code 16.

Args:

- registeraddress (int): The slave register start address (use decimal numbers, not hex).
- textstring (str): The string to store in the slave
- numberOfRegisters (int): The number of registers allocated for the string.

If the textstring is longer than the 2*numberOfRegisters, an error is raised. Shorter strings are padded with spaces.

Returns: None

Raises: ValueError, TypeError, IOError

`minimalmodbus.`**`PARITY`** **= 'N'**
Default value for the parity. See the pySerial module for documentation. Defaults to serial.PARITY_NONE

`minimalmodbus.`**`STOPBITS`** **= 1**
Default value for the number of stopbits (int).

`minimalmodbus.`**`TIMEOUT`** **= 0.05**
Default value for the timeout value in seconds (float).

2.7 stepperMotor module

class `stepperMotor.`**`StepperMotor`**(*name*, *adress*, *master*, *serialPort='/dev/ttyUSB0'*, *baudrate=19200*, *stopbits=1*, *parity='N'*, *timeout=0.05*, *standardWaitTime=0.02*, *waitForPingTime=0.2*, *maxFailCounter=50*)
Bases: *`minimalmodbus.Instrument`*, `object`

An Oriental Motor stepper motor. This class uses minimalmodbus to communicate over Modbus/RS-485 with the motor.

Note: Oriental Motors uses upper and lower registers (each 16 bit). Only lower registers are supported in this class. If you have to store values greater than 2**16 you have to use upper registers too.

Attributes:

- baudrate: The baudrate of the serial communication
- stopbits: Stopbit configuration of the serial communication

- parity: Parity configuration of the serial communication
- timeout: Modbus timeout
- waitForPingTime: Time between pings to the motor to check if its still moving
- standardWaitTime: Time to wait after each modbus communication
- maxFailCounter: If a communication failes maxFailCounter times a exception is raised
- master: An instance of the class which is controlling the motor. Must have an attribute named

"isInterrupted" and a method named "handleInterrupt" * registers: See oriental motor documentation

`__init__`(*name*, *adress*, *master*, *serialPort='/dev/ttyUSB0'*, *baudrate=19200*, *stopbits=1*, *parity='N'*, *timeout=0.05*, *standardWaitTime=0.02*, *waitForPingTime=0.2*, *maxFailCounter=50*)

Contructor. Initializes everything. See Attributes.

Args:

- name: The stepper motor name
- adress: The modbus adress of the motor
- serialport: The usb serial port which the motor is attached to
- rest: See class Attributes

Returns: None

Raises: None

`getBitFromRegister`(*adress*, *bit*)

Reads a bit from a register.

Args:

- adress (int): Register adress
- bit (int): Number of the bit

Returns: True or False

Raises: None

`getStatus`(*message*)

Args:

- message: The massage to wrap

Returns: A wrapped string containing name and message

Raises: None

`goForward`()

Starts forward moving of the motor. Args:

None

Returns: None

Raises: None

`goHome`()

Drives the motor to home position. Args:

None

Returns: None

Raises: None

`goReverse`()

Starts reverse moving of the motor. Args:

None

Returns: None

Raises: None

`readRegisterSafe`(*adress*)

Reads a value from a register making sure the communication was successfull. Raises an exception if the communication failes maxFailCounter times.

Args:

- adress (int): The adress to from

Returns: None

Raises: IOError

`startOperation`(*operationNumber*)

Starts a operation which is stored in the operation registers.

Args:

- operationNumber (int): The operation number to start

Returns: None

Raises: ValueError if the operation number is invalid

`stopMoving`()

Stops the motor.

Args: None

Returns: None

Raises: None

`waitFor`()

Waits for the motor to finish the operation.

Args: None

Returns: None

Raises: None

`writeOperationMode`(*operationMode*, *operationNumber*)

Writes the operation mode for a operation number.

Args:

- operationMode (0|1): The operation mode for the operation. 0 for incremental,

1 for absolute * operationNumber (int)

Returns: None

Raises: ValueError

`writeOperationPosition`(*operationPosition*, *operationNumber*)
Writes the operation position for a operation number.

Args:

- operationPosition (int): The steps to move for the operation number
- operationNumber (int)

Returns: None

Raises: ValueError

`writeOperationSpeed`(*operationSpeed*, *operationNumber*)
Writes the operation speed for a operation number.

Args:

- operationSpeed (int): The speed in Hz the operation is performed
- operationNumber (int)

Returns: None

Raises: ValueError

`writeRegisterSafe`(*adress*, *value*)
Writes a value to a register making sure the communication was successfull. Raises an exception if the communication failes maxFailCounter times.

Args:

- adress (int): The adress to write to
- value (int): The value to write

Returns: None

Raises: IOError

`writeToInputRegister`(*value*)
Writes a value to the input register and resets it.

Args: None

Returns: None

Raises: None

PYTHON MODULE INDEX