Programming manual for stepper motor positioning controls

Valid for firmware version 10.10.2009

NANOTEC ELECTRONIC GmbH & Co. KG
Gewerbestraße 11
D-85652 Landsham near Munich, Germany

Tel. +49 (0)89-900 686-0
Fax +49 (0)89-900 686-50
info@nanotec.com

Editorial

Nanotec® Electronic GmbH & Co. KG
Gewerbestraße 11
D-85652 Landsham / Pliening, Germany

Tel.: +49 (0)89-900 686-0
Fax: +49 (0)89-900 686-50

Internet: www.nanotec.com

Version/Change overview

Version	Date	Changes
V1.0	2009-02-10	Command reference created (firmware version 04.12.2008)
V2.0	2009-12-11	Instruction sets (firmware version 10.10.2009), supplemented by Java programming and COM interface programming, hence renamed as “Programming Manual”
V2.1	2010-01-28	Instruction sets
V2.2	2010-02-11	Jerkfree ramp instruction set
V2.3	2010-04-08	New notice: Java program and serial communication possible at the same time

Contents

1 About this manual

Target group

This technical manual is aimed at programmers who wish to program their own controller software for communication with controllers for the following Nanotec motors:

- SMCI12 (available from 02/10)
- SMCI33 *
- SMCI35 (available from 12/09)
- SMCI47-S *
- SMCP33
- PD4-N (available from 12/09)
- PD6-N

* Please note following information!

Information on SMCI33 and SMCI47-S

For drivers with firmware more recent than 30.04.2009 the update to the new firmware that is described in this manual cannot be carried out by the customer.

Please send us these drivers, we will carry out the update for you quickly and, of course, free of charge.

Contents of the manual

This manual contains a reference to all commands for controlling Nanotec motors (Chapter 2). Chapter 3 describes how to program them with Java (NanoJEasy), Chapter 4 describes how to program them via the COM interface.

Please note!

This programming manual must be read carefully before the Nanotec firmware command references are used for creating controller programs.

In the interests of its customers and to improve the function of this product Nanotec reserves the right to make technical alterations and further develop hardware and software without prior notice.

This manual was created with due care. It is exclusively intended as a technical description of the Nanotec firmware command references and the programming by JAVA or the COM interface. The warranty is limited to the repair or replacement of defective equipment of the Nanotec stepper motors, according to our general terms and conditions; liability for damage or errors resulting from the incorrect use of the command references for the programming of motor drivers is excluded.

For criticisms, proposals and suggestions for improvement, please contact the address given in the Editorial (page 2) or send an email to: info@nanotec.com

2 Command reference of the Nanotec firmware

2.1 General information

2.1.1 Command structure

Controller command structure

A command begins with the start character '#' and ends with a carriage return '\r'. All characters in between are ASCII characters (i.e. they are not control characters).

The start character is followed by the address of the motor as an ASCII decimal number.
This value may be from 1 to 254. If '*' is sent instead of the number, all drivers connected to the bus are addressed.

This is followed by the actual command which generally consists of an ASCII character and an optional ASCII number. This number must be written in decimal notation with a prefix of '+' or '-'.

When the user sends a setting to the firmware, a '+' sign is not mandatory for positive numbers.

Note:
Some commands consist of multiple characters while others do not require a number as a parameter.

Controller response

If a controller recognizes a command as relevant to it, it confirms receipt by returning the command as an echo but without the '#' start character.

If the controller receives an unknown command, it responds by returning the command followed by a question mark '?'.

The response of the controller ends with carriage return '\r', like the command itself. The address is returned as '001' and not as '1'.

If invalid values are transmitted to the controller, these are ignored but sent back as an echo anyway.

Example

Value transmitted to the controller:	“#1G=10000000\r”
Firmware response: ignored)	“001G10000000\r” (value will be

Examples

Setting the travel distance of controller 1:"#1s1000\r" -> "001s1000\r"

Starting a record:	“#1A\r” -> “001A\r”
Invalid command:	“#1°\r” -> “001°?\r”

CanOpen interface specification

Information on programming with CanOpen can be found in the corresponding manual for the interface at www.nanotec.com.

2.1.2 Long command format

Use

With the launch of the new firmware, commands were introduced that consist of more than one character. These commands are used for reading and changing machine parameters. Because these usually only have to be set during startup, the slower transmission speed due to the length of the command has no effect on operation.

Long command structure

A long command begins with the addressing scheme already described ("#<ID>"). This is followed by a colon that marks the beginning of the long command. Next comes the keyword and the command, followed by a carriage return character ("\r") that indicates the end of the command.

A long command can consist of the characters "A" to "Z" or "a" to "z" and the underscore ("_"). The syntax is case sensitive. Digits are not allowed.

Keywords

The following keywords are defined for long commands:

:CL For the controller settings and the motor settings (closed loop)

:brake For the motor controller

:Capt For the scope mode

Controller response

The firmware response does not begin with a "#" like the user request.

If the values are positive, the keyword is followed by a "+" sign. For negative values, a "-" sign is used.

Both signs ("+" and "-") can be used as separators.

If an unknown keyword is sent (unknown command), the firmware responds with a question mark after the colon.

Example

Unknown command: “#ID:CL_does_not_exist\r”

Firmware response: “ID:?\r"

Command for reading a parameter

Read command

To read a parameter, the end of the command name is terminated with a carriage return character.

Read command: "#ID:keyword_command_abc\r"

Firmware response

The firmware responds with an echo of the command and its value.

Response: "ID:keyword_command_abc+value\r"

Command for changing a parameter

Change command

To change a parameter, the command name is followed by a "=" character, followed by the value to be set. For positive values, a "+" sign is not mandatory but is also not disallowed. The command is terminated with a carriage return character.

Change command: "#ID:keyword_command_abc=value\r"

Firmware response

The firmware responds with an echo of the command as confirmation.

Response: "ID:keyword_command_abc+value\r"

See also the following example.

Example

The structure of the long command is shown in the following example:

"Read out the motor pole pairs"

Read parameter	"#1:CL_motor_pp\r"
Firmware response	"1:CL_motor_pp+50\r"
Change parameter	"#1:CL_motor_pp=100\r"
Firmware response	"1:CL_motor_pp100\r"

2.2 Command overview

Below you will find an overview of the serial commands of the Nanotec firmware (characters and parameters):

2.3 Read command

Function

A series of settings that can be set with a specific command can be read out with a corresponding read command.

Command

Character	Parameter
'Z' + parameter '	The read command is composed of the 'Z' character and the command for the corresponding parameter.

Example

Read out the travel distance: "#1Zs\r" -> "001Zs1000\r"

Firmware response

Returns the required parameter.

Description

This is used to read out the current settings of the values of certain commands. For example, the travel distance is read out with 'Zs' to which the firmware responds with 'Zs1000'.

If the parameter of a specific record is to be read out, the number of the record must be placed in front of the respective command.

Example: 'Z5s' -> 'Z5s2000'

A list of record commands can be found under "*2.4 Records*"

2.4 Records

Saving travel distances

The firmware supports the saving of travel distances in records. These data are saved in an EEPROM and, consequently, are retained even if the device is switched off.

The EEPROM can accommodate 32 records with record numbers 1 to 32.

Saved settings per record

The following settings are saved in every record:

Setting	Para-meter	See section	Page
Position mode	'p'	*2.6.6 Setting positioning* mode	43
Travel distance	's'	*2.6.8 Setting the* travel distance	46
Initial step frequency	'u'	*2.6.9 Setting the* minimum frequency	47
Maximum step frequency	'o'	*2.6.10 Setting the* maximum frequency	48
Second maximum step frequency	'n'	*2.6.11 Setting the maximum* frequency 2	48
Acceleration and braking ramp	'b'	*2.6.12 Setting the* acceleration ramp	49
Direction of rotation	'd'	*2.6.15 Setting the* direction of rotation	50
Reversal of direction of rotation for repeat records	't'	*2.6.16 Setting the* change of direction	51
Repetitions	'W'	*2.6.17 Setting the* repetitions	51
Pause between repetitions and continuation records	'P'	*2.6.18 Setting the* record pause	52
Record number of continuation record	'N'	*2.6.19 Setting the* continuation record	52

2.5 General commands

2.5.1 Setting the phase current

Parameter

Character	Permissible values	Writable	Data type	Default value
'i'	0 to 150	Yes	u8 (integer)	depending on controller

Firmware response

Confirms the command through an echo.

Description

Sets the phase current in percent. Values above 100 should be avoided.

Reading out

Command 'Zi' is used to read out the current valid value.

2.5.2 Setting the phase current at a standstill

Parameter

Character	Permissible values	Writable	Data type	Default value
'r'	0 to 150	Yes	u8 (integer)	depending on controller

Firmware response

Confirms the command through an echo.

Description

Sets the current of the current reduction in percent. Like the phase current, this current is relative to the end value and not relative to the phase current. Values above 100 should be avoided.

Reading out

Command 'Zr' is used to read out the current valid value.

2.5.3 Setting the step mode

Parameter

Character	Permissible values	Writable	Data type	Default value
'g'	0 to 255	Yes	u8 (integer)	2 = half step

Firmware response

Confirms the command through an echo.

Description

Sets the step mode. The number handed over equals the number of microsteps per full step, with the exception of the value 255 which selects the adaptive step mode.

Reading out

Command 'Zg' is used to read out the current valid value.

2.5.4 Setting the motor address

Parameter

Character	Permissible values	Writable	Data type	Default value
'm'	1 to 254	Yes	u8 (integer)	1

Firmware response

Confirms the command through an echo.

Description

Sets the motor address. Ensure that the newly set address is not already occupied by another motor as this would make communication impossible.

Addresses 0 and 255 are reserved for faults of the EEPROM.

Reading out

Command 'Zm' is used to read out the current address. See also command *2.5.20 Reading out* the motor address *'M'*.

2.5.5 Setting the motor mode

Parameter

Character	Permissible values	Writable	Data type	Default value
'!'	1 to 101	Yes	u8 (integer)	1

Firmware response

Confirms the command through an echo.

If invalid values are set for motor mode '!' and positioning mode 'p', the motor mode '!' retains its value and the positioning mode 'p' is set to 1.

Description

Sets the motor mode. The following modes are available:

For old scheme:

1: Positioning mode

2: Speed mode

3: Flag positioning mode

4: Clock direction mode

5: Analog mode

6: Joystick mode

7: Analog positioning mode

8: HW reference mode

9: Torque mode

101: CL quick test mode

101: CL test mode

For more information, see command *2.6.6 Setting positioning mode (old* scheme) *'p'*.

For new scheme:

10: Motor mode

For more information, see command *2.6.7 Setting the positioning mode* (new scheme) *'p'*.

Reading out

Command 'Z!' is used to read out the current valid value.

2.5.6 Setting the limit switch behavior

Parameter

Character	Permissible values	Writable	Data type	Default value
'l'	0 to 4294967295	Yes	u32 (integer)	17442

Firmware response

Confirms the command through an echo.

Description

Sets the limit switch behavior. The integer parameter is interpreted as a bit mask. The bit mask has 16 bits.

"Free travel" means that, when the switch is reached, the controller travels away from the switch at the set lower speed.

"Stop" means that, when the switch is reached, the controller stops immediately. The switch remains pressed.

Behavior of the internal limit switch during a reference run:

Bit0: Free travel forwards
Bit1: Free travel backwards (default value)
Exactly one of the two bits must be set.

Behavior of the internal limit switch during a normal run:

Bit2: Free travel forwards
Bit3: Free travel backwards
Bit4: Stop
Bit5: Ignore (default value)
Exactly one of the four bits must be set.
This setting is useful when the motor is not allowed to turn more than one rotation.

Behavior of the external limit switch during a reference run:

Bit9: Free travel forwards
Bit10: Free travel backwards (default value)
Exactly one of the two bits must be set.

Behavior of the external limit switch during a normal run:

Bit11: Free travel forwards
Bit12: Free travel backwards
Bit13: Stop
Bit14: Ignore (default value)
Exactly one of the four bits must be set.
With this setting, the travel distance of the motor can be precisely limited by a limit switch.

Reading out

Command 'Zl' is used to read out the current valid value.

2.5.7 Setting the limit switch type

Parameter

Character	Permissible values	Writable	Data type	Default value
'e'	0 and 1	Yes	u8 (integer)	0

Firmware response

Confirms the command through an echo.

Description

Specifies the type of limit switch:

- Value 0: Opener
- Value 1: Closer

This parameter is used to indicate to the firmware the state in which it sees the external limit switch as activated. The limit switch is connected between the supply voltage (to +5V in SMCIxx) and input 6.

Therefore, 'opener' means that, under normal conditions, a high level is applied at the input since the switch is normally closed. When the switch is activated, it opens this contact ("opener") and there is no voltage at the input.

Reading out

Command 'Ze' is used to read out the current valid value.

2.5.8 Setting the step angle

Parameter

Character	Permissible values	Writable	Data type	Default value
'a'	0 to 255	Yes	u8 (integer)	18

Firmware response

Confirms the command through an echo.

Description

To convert the encoder position to the rotor position, the firmware requires information about the step angle of the motor. A value of 9 must be set for 0.9° motors, and 18 must be set for 1.8° motors. Other values are not supported.

Reading out

Command 'Za' is used to read out the current setting of the value.

2.5.9 Setting the error correction mode

Parameter

Character	Permissible values	Writable	Data type	Default value
'U'	0 and 1	Yes	u8 (integer)	0

Firmware response

Confirms the command through an echo.

Description

Sets the error correction mode:

- Value 0: Off
- Value 1: Correction after travel

In a motor without an encoder, this value must be explicitly set to 0; otherwise, it will continuously attempt to make a correction because it assumes that there are step losses.

Reading out

Command 'ZU' is used to read out the current setting of the value.

2.5.10 Setting the record for auto correction

Parameter

Character	Permissible values	Writable	Data type	Default value
'F'	0 to 32	Yes	u8 (integer)	0

Firmware response

Confirms the command through an echo.

Description

The ramp and the speed in the selected record (integer) are used for the correction run.
See command *2.5.9 Setting the* error correction mode *'U'*.

Reading out

Command 'ZF' is used to read out the current valid value.

2.5.11 Setting the encoder direction

Parameter

Character	Permissible values	Writable	Data type	Default value
'q'	0 and 1	Yes	u8 (integer)	0

Firmware response

Confirms the command through an echo.

Description

If the parameter is set to '1', the direction of the rotary encoder is reversed.

Reading out

Command 'Zq' is used to read out the current valid value.

2.5.12 Setting the settling time

Parameter

Character	Permissible values	Writable	Data type	Default value
'O'	0 to 250	Yes	u8 (integer)	8

Firmware response

Confirms the command through an echo.

Description

Defines the settling time in 10ms steps between the end of the run and when the position is checked by the encoder.

This parameter is only valid for the positional check after a run.
See command *2.5.9 Setting the* error correction mode *'U'*.

Between repetitions or continuation records, this position is only checked if the pause time (see command *2.6.18 Setting the* record pause *'P'*) is longer than the swing out time.

After a record, the settling time is awaited before the motor indicates that it is ready again.

Reading out

Command 'ZO' is used to read out the current valid value.

2.5.13 Setting the maximum encoder deviation

Parameter

Character	Permissible values	Writable	Data type	Default value
'X'	0 to 250	Yes	u8 (integer)	2

Firmware response

Confirms the command through an echo.

Description

Specifies the maximum deviation in steps between the setpoint position and the encoder position.

In step modes greater than 1/1 step in 10° and 1/1 step in 5° motors, this value must be greater than 0 since, even then, the encoder has a lower resolution than the microsteps of the motor.

Reading out

Command 'ZX' is used to read out the current valid value.

2.5.14 Resetting the position error

Parameter

Character	Permissible values	Writable	Data type	Default value
'D'	-2147483648 to +2147483647	Yes	s32 (integer)	0

Firmware response

Confirms the command through an echo.

Description

Resets an error in the speed monitoring and sets the current position to the position indicated by the encoder (for input without parameters, C is set to I, see section *2.5.15* and *2.5.16*).

For input with parameters, C and I are set to the parameter value.
Example: 'D100' -> C=100; I=100

2.5.15 Reading out the error memory

Parameter

Character	Permissible values	Writable	Data type	Default value
'E'	–	No	–	–

Firmware response

Returns the index of the error memory with the last error that occurred.

Description

The firmware contains 32 error memory locations.

The last 32 errors are stored. When memory location 32 is reached, the next error is again stored at memory position 1. In this case, memory position 2 contains the oldest error code that can be read out.

This command is used to read out the index of the memory space with the last error that occurred and the corresponding error code.

Reading out

Command 'Z' + Index number + 'E' is used to read out the error number of the respective error memory.
Example: 'Z32E' returns the error number of index 32.

Error codes

```
//! Error codes for error byte in EEPROM
#define ERROR_LOWVOLTAGE    0x01
#define ERROR_TEMP          0x02
#define ERROR_TMC           0x04
#define ERROR_EE            0x08
#define ERROR_QEI           0x10
#define ERROR_INTERNAL      0x20
```

Meaning

Error	Meaning
LOWVOLTAGE	Undervoltage
TMC	Controller module returned one error.
EE	Incorrect data in EPROM, e.g. step resolution is 25th of one step.
QEI	Position error
INTERNAL	Internal error (equivalent to the Windows blue screen).

Controller status

The status of the controller can be read out with the *2.5.21 Reading out* the status *'$'* command.

2.5.16 Reading out the encoder position

Parameter

Character	Permissible values	Writable	Data type	Default value
'I'	–	No	–	–

Firmware response

Returns the current position of the motor according to the encoder.

Description

In motors with an encoder, this command returns the current position of the motor in motor steps as indicated by the encoder. Provided that the motor has not lost any steps, the values of the *2.5.17 Reading out* the position *'C'* command and the *2.6.4 Reading out the* current record *' |'* (pipe) command are the same.

However, it should be noted that the encoder has a resolution that is too low for step modes greater than 1/1 in 10° motors and 1/1 in 5° motors, and differences will therefore still arise between the two values specified above.

2.5.17 Reading out the position

Parameter

Character	Permissible values	Writable	Data type	Default value
'C'	–	No	–	–

Firmware response

Returns the current position.

Description

Returns the current position of the motor in steps of the set step mode. This position is relative to the position of the last reference run.

If the motor is equipped with an angle transmitter, this value should be very close to the value of command "I" with a very low tolerance.

The tolerance depends on the step mode and the motor type (0.9° or 1.8°) since the angle transmitter has a lower resolution than the motor in the microstep mode.

The value range is that of a 32-bit signed integer (range of values ± 2^{31}).

2.5.18 Resetting the position

Parameter

Character	Permissible values	Writable	Data type	Default value
'c'	–	No	–	–

Firmware response

Confirms the command through an echo.

Description

Resets the position of the motor to 0.

The current position of the motor is then used as the reference position.

2.5.19 Request “Motor is referenced”

Parameters

Character	Permissible values	Writable	Data type	Default value
':is_referenced'	0 and 1	No	u8 (integer)	0

Firmware response

If the motor has already been referenced, “1” is returned, otherwise “0”.

Description

Parameter is '1' after the reference run.

See also *2.5.14 Resetting the* position error and *2.5.18 Resetting the* position.

2.5.20 Reading out the motor address

Parameter

Character	Permissible values	Writable	Data type	Default value
'M'	–	No	–	–

Firmware response

Returns the motor address.

Description

Returns the serial address. In particular, this is useful in connection with the **'*'** addressing type if the motor address is not known.

2.5.21 Reading out the status

Parameter

Character	Permissible values	Writable	Data type	Default value
'$'	–	No	–	–

Firmware response

Returns the status of the firmware as a bit mask.

Description

The bit mask has 8 bits.

Bit 0: 1: Controller ready

Bit 1: 1: Zero position reached

Bit 2: 1: Position error

Bit 3: 1: Input 1 is set while the controller is ready again. This occurs when the controller is started via input 1 and the controller is ready before the input has been reset.

Bits 4 through 6 specify the current mode as an integer:

0: Unused

1: Controller in positioning mode

2: Controller in speed mode

3: Controller in flag positioning mode

4: Controller in clock direction mode

5: Analogue mode

6: Joystick mode

7: Unused

Bit 7 is unassigned

2.5.22 Reading out the firmware version

Parameter

Character	Permissible values	Writable	Data type	Default value
'v'	–	No	–	–

Firmware response

Returns the version string of the firmware.

Description

The return sting consists of several blocks:

'v' echo of the command

' ' separator (space)

Hardware: 'PD4','PD4lc','PD2lc','SMCI32','SMCI47' are possible versions

'_' separator

Communication: 'USB' or 'RS485'

'_' separator

Release date: dd-mm-yyyy, e.g. 26-09-2007

Example of a complete response

"001v PD4_RS485_26-09-2007\r"

2.5.23 Reading out the firmware version (old)

Parameter

Character	Permissible values	Writable	Data type	Default value
' ' (space)	–	No	–	–

Firmware response

String containing firmware version (const, since new command 'v' has assumed this function).

Description

Required for bootloader; otherwise, this command serves no purpose.

2.5.24 Masking and demasking the inputs

Parameters

Character	Permissible values	Writable	Data type	Default value
'L'	0 to 4294967295	Yes	u32 (integer)	0x0003003f

Firmware response

Confirms the command through an echo.

Invalid values are ignored, i.e. the entire mask is discarded.

Description

This bit mask has 32 bits.

Sets a bit mask that permits the user to use the inputs and outputs. If the bit of the corresponding I/Os is set to '1' , the firmware uses these I/Os. If it is set to '0', the I/Os are available to the user. See also command *2.5.27 Setting the* outputs *'Y'*.

The bit assignment is shown below:	Bit on 1:
Bit0: Input 1	1
Bit1: Input 2	2
Bit2: Input 3	4
Bit3: Input 4	8
Bit4: Input 5	16
Bit5: Input 6	32
Bit16: Output 1	65536
Bit17: Output 2	131072
All other bits are '0'	All on 1: 196671

Attention:
If a bit is not addressed when the mask is set, it is automatically set to '0', regardless of the state. All bits must be set at once.

If invalid bit masks are used, these are discarded, even if the firmware confirms them correctly.

Reading out

Command 'ZL' is used to read out the current setting of the mask.

Examples

All bits should be set to '0'.

Send: #1L0\r

Read: 1L0\r

Bit3 and Bit5 should be set to '1':

Send: #1L20\r

Read: 1L20\r

'20' because Bit3 is addressed with the value of 4 and Bit5 with the value of 16, i.e. 4 + 16 = 20.

2.5.25 Reversing the polarity of the inputs and outputs

Parameters

Character	Permissible values	Writable	Data type	Default value
'h'	0 to 4294967295	Yes	u32 (integer)	0x0003003f

Firmware response

Confirms the command through an echo.

Invalid values are ignored, i.e. the entire mask is discarded.

Description

Sets a bit mask with which the user can reverse the polarity of the inputs and outputs. If the bit of the corresponding I/O is set to '1', there is no polarity reversal. If it is set to '0', the polarity of the I/O is inverted.

The bit assignment is shown below:

Bit0: Input 1

Bit1: Input 2

Bit2: Input 3

Bit3: Input 4

Bit4: Input 5

Bit5: Input 6

Bit16: Output 1

Bit17: Output 2

All other bits are '0'.

If invalid bit masks are used, these are discarded, even if the firmware confirms them correctly.

Reading out

Command 'Zh' is used to read out the current setting of the mask.

2.5.26 Setting the debounce time for the inputs

Parameters

Character	Permissible values	Writable	Data type	Default value
'K'	0 to 20	Yes	u8 (integer)	20

Firmware response

Confirms the command through an echo.

Description

Sets the time in ms that needs to elapse after a signal change at an input until the signal has stabilized (so-called "debouncing").

Reading out

Command 'ZK' is used to read out the current setting of the value.

2.5.27 Setting the outputs

Parameters

Character	Permissible values	Writable	Data type	Default value
'Y'	0 to 4294967295	Yes	u32 (integer)	0

Firmware response

Confirms the command through an echo.

Description

This bit mask has 32 bits.

Sets the outputs of the firmware, provided that these have been masked for free use using the *2.5.24 Masking and demasking the* inputs *'L'* command.

Output 1 corresponds to bit 16 and output 2 to bit 17.

Reading out

Command 'ZY' is used to read out the current setting of the value.

The status of the inputs is displayed as well.

Bit0: Input 1

Bit1: Input 2

Bit2: Input 3

Bit3: Input 4

Bit4: Input 5

Bit5: Input 6

Bit6: '0' when the encoder is at the index line, otherwise '1'

Bit 16: Output 1 (as set by the user, even if the firmware is currently using it)

Bit 17: Output 2 (as set by the user, even if the firmware is currently using it)

All other bits are '0'.

2.5.28 Carrying out an EEPROM reset

Parameters

Character	Permissible values	Writable	Data type	Default value
'~'	–	Yes	–	–

Firmware response

Confirms the command through an echo.

Description

Restores the the factory defaults again. The controller requires a second until new commands are accepted.

2.5.29 Setting automatic sending of the status

Parameter

Character	Permissible values	Writable	Data type	Default value
'J'	0 and 1	Yes	u8 (integer)	0

Firmware response

Confirms the command through an echo.

Description

If this parameter is set to '1', the firmware independently sends the status after the end of a run. See command *2.5.21 Reading out* the status *'$'*, with the difference that a lower case 'j' is sent instead of the '$'.

Reading out

Command 'ZJ' is used to read out the current valid value.

2.5.30 Starting the bootloader

Parameter

Character	Permissible values	Writable	Data type	Default value
'@S'	–	Yes	–	–

Firmware response

No response, bootloader responds with '@OK'

Description

The command instructs the firmware to launch the bootloader. The firmware itself does not respond to the command. The bootloader responds with '@OK'.

The bootloader itself also requires this command to prevent it from automatically terminating itself after one half second. Therefore, this command needs to be sent repeatedly until the bootloader responds with '@OK'. The bootloader uses the same addressing scheme as the firmware itself.

2.5.31 Setting the reverse clearance

Parameter

Character	Permissible values	Writable	Data type	Default value
'z'	0 to 9999	Yes	u16 (integer)	0

Firmware response

Confirms the command through an echo.

Description

Specifies the reverse clearance in steps.

This setting is used to compensate for the clearance of downstream gears when there is a change in direction.

When there is a change in direction, the motor takes the number of steps set in the parameter before it begins incrementing the position.

Reading out

Command 'Zz' is used to read out the current valid value.

2.5.32 Setting the ramp

Parameters

Character	Permissible values	Writable	Data type	Default value
':ramp_mode'	0, 1 and 2	Yes	u16 (integer)	0

Firmware response

Confirms the command through an echo.

Description

Sets the ramp in all modes:

- "0" = The trapezoid ramp is selected
- "1" = The sinusoidal ramp is selected
- "2" = The jerkfree ramp is selected

This parameter applies for all modes except clock direction and current mode (as these modes do not generally use a ramp).

Reading out

If the keyword is sent without a “= + value”, the current setting of the value can be read out.

2.5.33 Setting the maximum jerk for the acceleration ramp

Parameters

Character	Permissible values	Writable	Data type	Default value
':b'	1 - 100000000	Yes	u32 (integer)	1

Firmware response

Confirms the command through an echo.

Description

Sets the maximum jerk for the acceleration.

Reading out

If the keyword is sent without a "= + value", the current setting of the value can be read out.

Note

The actual ramp results from the values for "b" and ":b".

- "b" = maximum acceleration
- ":b" = maximum change of the acceleration (max. jerk)

2.5.34 Setting the maximum jerk for the braking ramp

Parameter

Character	Permissible values	Writable	Data type	Default value
':B'	1 - 100000000	Yes	u32 (integer)	0

Firmware response

Confirms the command through an echo.

Description

Sets the maximum jerk for the braking ramp.

If the value is set to "0", the same value is used for braking as for accelerating (":b").

Reading out

If the keyword is sent without a "= + value", the current setting of the value can be read out.

Note

The actual ramp results from the values for "B" and ":B".

- "B" = maximum acceleration
- ":B" = maximum change of the acceleration (max. jerk)

2.5.35 Setting the wait time for switching off the brake voltage

Parameters

Character	Permissible values	Writable	Data type	Default value
':brake_ta'	0 to 65535	Yes	u16 (integer)	0

Unit

ms

Firmware response

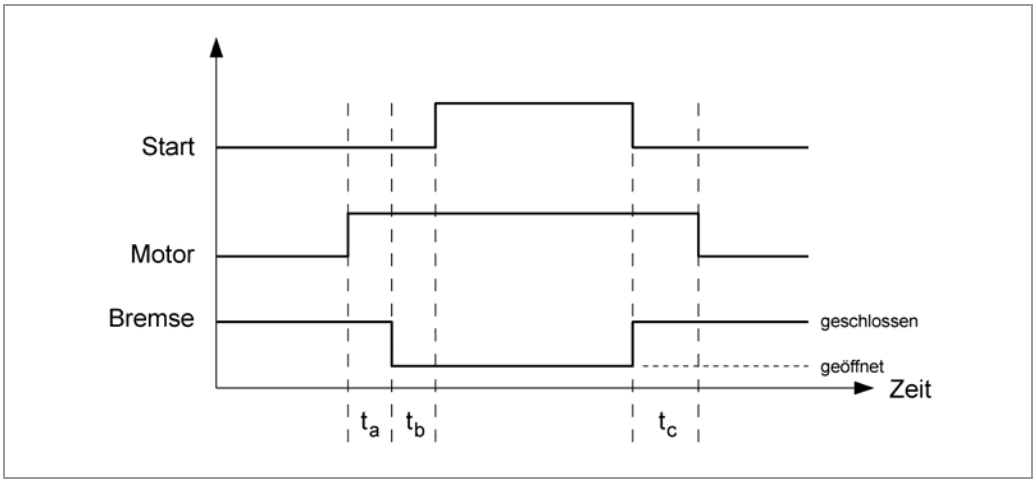
Confirms the command through an echo.

Description

The external brake can be set via the following parameters:

- Time ta:
  Waiting time between switching on the motor current and switching off (triggering) the brake in milliseconds.
- Time tb:
  Waiting time between switching off (triggering) the brake and activation of readiness in milliseconds. Travel commands will only be executed after this waiting time.
- Time tc:
  Waiting time between switching on the brake and switching off the motor current in milliseconds.

The parameters indicate times between 0 and 65,536 milliseconds.
Default values of the controller after a reset: 0 ms.



When switching on the controller, the brake becomes active first and the motor is not provided with power. First the motor current is switched on and a period of ta ms waited. Then the brake is disengaged and a period of tb ms waited. Travel commands will only be executed after expiry of ta and tb.

Note:
During current reduction, the brake is not actively connected.

Reading out

If the keyword is sent without a “= + value”, the current setting of the value can be read out.

2.5.36 Setting the wait time for the motor movement

Parameters

Character	Permissible values	Writable	Data type	Default value
':brake_tb'	0 to 65535	Yes	u16 (integer)	0

Unit

ms

Firmware response

Confirms the command through an echo.

Description

Sets the wait time in milliseconds between switching off of the brake voltage and enabling of a motor movement.

For more information, see command *2.5.35 Setting the wait time for switching off the brake voltage'ta'*.

Reading out

If the keyword is sent without a “= + value”, the current setting of the value can be read out.

2.5.37 Setting the wait time for switching off the motor current

Parameters

Character	Permissible values	Writable	Data type	Default value
':brake_tc'	0 to 65535	Yes	u16 (integer)	0

Unit

ms

Firmware response

Confirms the command through an echo.

Description

Sets the wait time in milliseconds between switching on of the brake voltage and switching off of the motor current.

For more information, see command *2.5.35 Setting the wait time for switching off the brake voltage'ta'*.

Reading out

If the keyword is sent without a “= + value”, the current setting of the value can be read out.

2.5.38 Setting baudrate of the controller

Parameters

Character	Permissible values	Writable	Data type	Default value
':baud'	0 to 255	Yes	u8 (integer)	12

Firmware response

Confirms the command through an echo.

Description

Sets the baudrate of the controller:

1 110
2 300
3 600
4 1200
5 2400
6 4800
7 9600
8 14400
9 19200
10 38400
11 57600
12 115200 (default value)

Note:
The new value is only activated (current off/on) after the controller is restarted.

Example

Command '#1:baud=8' is used to set the baudrate of the 1st. controller to 14400 baud.

Reading out

Command 'Z:baud' is used to read out the current valid value.

2.6 Record commands

2.6.1 Starting the motor

Parameters

Character	Permissible values	Writable	Data type	Default value
'A'	–	No	–	–

Firmware response

Confirms the command through an echo.

Description

Starts the run with the current parameter settings.

2.6.2 Stopping a motor

Parameters

Character	Permissible values	Writable	Data type	Default value
'S'	0 and 1	Yes	u8 (integer)	0

Firmware response

Confirms the command through an echo.

Description

Cancels the current travel. The following ramps are used:

- Quickstop (H) if there is no argument or the argument is "0"
- Brake ramp (B) if the argument is "1"

2.6.3 Loading a record from the EEPROM

Parameter

Character	Permissible values	Writable	Data type	Default value
'y'	1 to 32	Yes	u8 (integer)	1

Firmware response

Confirms the command through an echo.

Description

Loads the record data of the record passed in the parameter from the EEPROM.

See also command *2.6.5 Saving a* record *'>'*.

2.6.4 Reading out the current record

Parameters

Character	Permissible values	Writable	Data type	Default value
'|' (pipe)	0 and 1	Yes	u8 (integer)	1

Firmware response

Confirms the command through an echo when the parameter is set to '1'. This is the only response.

Description

If the parameter is set to '0', the firmware does not respond at all to commands, although it continues to execute them as before. This can be used to quickly send settings to the firmware without awaiting a response.

Reading out

With command 'Z|', the firmware sends all settings of the loaded record together.

With 'Z5|', the data of set 5 in the EEPROM are sent.

The format corresponds to that of the respective commands.

It should be noted that the '|' character is not sent with the response. See the following examples.

Examples

'#1Z|\r'

-> 'Zp+1s+1u+400o+860n+1000b+55800d+1t+0W+1P+0N+0\r'

'#1Z5|\r'

-> 'Z5p+1s+400u+400o+1000n+1000b+2364d+0t+0W+1P+0N+0\r'

2.6.5 Saving a record

Parameter

Character	Permissible values	Writable	Data type	Default value
'>'	1 to 32	Yes	u8 (integer)	1

Firmware response

Confirms the command through an echo.

Description

This command is used to save the currently set commands (in RAM) in a record in the EEPROM. The parameter is the record number in which the data are saved.

This command should not be called up during a run because the current values change during subsequent runs.

A record contains the following settings and commands:

Setting	Para-meter	See section	Page
Position mode	'p'	*2.6.6 Setting positioning* mode	43
Travel distance	's'	*2.6.8 Setting the* travel distance	46
Initial step frequency	'u'	*2.6.9 Setting the* minimum frequency	47
Maximum step frequency	'o'	*2.6.10 Setting the* maximum frequency	48
Second maximum step frequency	'n'	*2.6.11 Setting the maximum* frequency 2	48
Acceleration and braking ramp	'b'	*2.6.12 Setting the* acceleration ramp	49
Direction of rotation	'd'	*2.6.15 Setting the* direction of rotation	50
Reversal of direction of rotation for repeat records	't'	*2.6.16 Setting the* change of direction	51
Repetitions	'w'	*2.6.17 Setting the* repetitions	51
Pause between repetitions and continuation records	'P'	*2.6.18 Setting the* record pause	52
Record number of continuation record	'N'	*2.6.19 Setting the* continuation record	52

2.6.6 Setting positioning mode (old scheme)

Parameters

Character	Permissible values	Writable	Data type	Default value
'p'	1 to 17	Yes	u8 (integer)	1

Firmware response

Confirms the command through an echo.

If invalid values are set for motor mode '!' and positioning mode 'p', the motor mode '!' retains its value and the positioning mode 'p' is set to 1.

Description

The motor modes (command '!') and positioning modes (command 'p') can be selected either according to the old scheme or according to the new scheme, see Section *2.6.7 Setting the positioning mode* (new scheme).

The value combinations of the old scheme for motor mode '!' and positioning mode 'p' are:

Positioning mode (!=1)	
p=1	Relative positioning; The command *2.6.8 Setting the* travel distance 's' specifies the travel distance relative to the current position. The command *2.6.15 Setting the* direction of rotation 'd' specifies the direction. The parameter *2.6.8 Setting the* travel distance 's' must be positive.
p=2	Absolute positioning; Command *2.6.8 Setting the* travel distance *'s'* defines the target position relative to the reference position. Command *2.6.15 Setting the* direction of rotation *'d'* is ignored.
p=3	Internal reference run; The motor runs with the lower speed in the direction set in command *2.6.15 Setting the* direction of rotation *'d'* until it reaches the index line of the encoder. Then the motor runs a fixed number of steps to leave the index line again. For the direction of free travel, see command *2.5.6 Setting the* limit switch behavior *'l'*. This mode is only useful for motors with integrated and connected encoders.
p=4	External reference run; The motor runs at the highest speed in the direction set in command *2.6.15 Setting the* direction of rotation *'d'* until it reaches the limit switch. Then a free run is performed, depending on the setting. See command *2.5.6 Setting the* limit switch behavior *'l'*.
Speed mode (!=2)	
p=1	Speed mode; When the motor is started, the motor increases in speed to the maximum speed with the set ramp. Changes in the speed or direction of rotation are performed immediately with the set ramp without having to stop the motor first.
p=2	Not assigned
p=3	Internal reference run; see Positioning mode
p=4	External reference run; see Positioning mode

Flag positioning mode (!=3)	
p=1	Flag positioning mode; After starting, the motor runs up to the maximum speed. After arrival of the trigger event (command *2.7.9 Actuating the* trigger *'T'* or trigger input) the motor continues to travel the selected travel distance (command *2.6.8 Setting the* travel distance *'s'*) and changes its speed to the maximum speed 2 (command *2.6.11 Setting the maximum* frequency 2 *'n'*) for this purpose.
p=2	Not assigned
p=3	Internal reference run; see Positioning mode
p=4	External reference run; see Positioning mode
Clock direction mode (!=4)	
p=1	Manual left.
p=2	Manual right.
p=3	Internal reference run; see Positioning mode
p=4	External reference run; see Positioning mode
Analog mode (!=5)	
	Not applicable
Joystick mode (!=6)	
	Not applicable
Analog positioning mode (!=7)	
p=1	Analog positioning mode
p=2	Not assigned
p=3	Internal reference run; see Positioning mode
p=4	External reference run; see Positioning mode
HW reference mode (!=8)	
	Not applicable
Torque mode (!=9)	
	Not applicable
CL quick test mode (!=101)	
p=1	CL quick test mode
CL test mode (!=101)	
p=2	CL test mode

Reading out

Command 'Zp' is used to read out the current valid value.

2.6.7 Setting the positioning mode (new scheme)

Parameters

Character	Permissible values	Writable	Data type	Default value
'p'	1 to 17	Yes	u8 (integer)	1

Firmware response

Confirms the command through an echo.

If invalid values are set for motor mode '!' and positioning mode 'p', the motor mode '!' retains its value and the positioning mode 'p' is set to 1.

Description

The motor modes (command '!') and positioning modes (command 'p') can be selected either according to the old scheme or according to the new scheme, see Section *2.6.6 Setting positioning mode (old* scheme).

In the new scheme all motor modes and positioning modes of the old scheme run under the same motor mode 10 ('!10') and the respective positioning mode ('p1' to 'p17'). This means the motor mode and the positioning mode can be saved in the records.

The value combinations of the new scheme for motor mode '!' and positioning mode 'p' are:

Positioning mode (!=10)	
p=1	Relative positioning; The command *2.6.8 Setting the* travel distance 's' specifies the travel distance relative to the current position. The command *2.6.15 Setting the* direction of rotation 'd' specifies the direction. The parameter *2.6.8 Setting the* travel distance 's' must be positive.
p=2	Absolute positioning; Command *2.6.8 Setting the* travel distance *'s'* defines the target position relative to the reference position. Command *2.6.15 Setting the* direction of rotation *'d'* is ignored.
p=3	Internal reference run; The motor runs with the lower speed in the direction set in command *2.6.15 Setting the* direction of rotation *'d'* until it reaches the index line of the encoder. Then the motor runs a fixed number of steps to leave the index line again. For the direction of free travel, see command *2.5.6 Setting the* limit switch behavior *'l'*. This mode is only useful for motors with integrated and connected encoders.
p=4	External reference run; The motor runs at the highest speed in the direction set in command *2.6.15 Setting the* direction of rotation *'d'* until it reaches the limit switch. Then a free run is performed, depending on the setting. See command *2.5.6 Setting the* limit switch behavior *'l'*.
Speed mode (!=10)	
p=5	Speed mode; When the motor is started, the motor increases in speed to the maximum speed with the set ramp. Changes in the speed or direction of rotation are performed immediately with the set ramp without having to stop the motor first.
p=3	Internal reference run; see Positioning mode

p=4	External reference run; see Positioning mode
Flag positioning mode (!=10)	
p=6	Flag positioning mode; After starting, the motor runs up to the maximum speed. After arrival of the trigger event (command *2.7.9 Actuating the* trigger *'T'* or trigger input) the motor continues to travel the selected travel distance (command *2.6.8 Setting the* travel distance *'s'*) and changes its speed to the maximum speed 2 (command *2.6.11 Setting the maximum* frequency 2 *'n'*) for this purpose.
p=3	Internal reference run; see Positioning mode
p=4	External reference run; see Positioning mode
Clock direction mode (!=10)	
p=7	Manual left.
p=8	Manual right.
p=9	Internal reference run; see Positioning mode
p=10	External reference run; see Positioning mode
Analog mode (!=10)	
p=11	Analog mode
Joystick mode (!=10)	
p=12	Joystick mode
Analog positioning mode (!=10)	
p=13	Analog positioning mode
p=3	Internal reference run; see Positioning mode
p=4	External reference run; see Positioning mode
HW reference mode (!=10)	
p=14	HW reference mode
Torque mode (!=10)	
p=15	Torque mode
CL quick test mode (!=10)	
p=16	CL quick test mode
CL test mode (!=10)	
p=17	CL test mode

Reading out

Command 'Zp' is used to read out the current valid value.

2.6.8 Setting the travel distance

Parameter

Character	Permissible	Writable	Data type	Default value

	values			
's'	-2147483648 to +2147483647	Yes	s32 (integer)	0

Firmware response

Confirms the command through an echo.

Description

This command specifies the travel distance in (micro-)steps. Only positive values are allowed for the relative positioning. The direction is set with command *2.6.15 Setting the* direction of rotation *'d'*.

For absolute positioning, this command specifies the target position. Negative values are allowed in this case. The direction of rotation from command *2.6.15 Setting the* direction of rotation '*d*' is ignored since it can be derived from the current position and the target position.

The value range is that of a 32-bit signed integer (range of values ± 2^{31}).

In the adaptive mode, this parameter refers to full steps.

Reading out

Command 'Zs' is used to read out the current valid value.

2.6.9 Setting the minimum frequency

Parameter

Character	Permissible values	Writable	Data type	Default value
'u'	1 to 160000	Yes	u32 (integer)	1

Firmware response

Confirms the command through an echo.

Description

Specifies the minimum speed in Hertz (steps per second).

When a record starts, the motor begins rotating with the minimum speed. It then accelerates with the set ramp (command *2.6.12 Setting the* acceleration ramp *'b'*) to the maximum speed (command *2.6.10 Setting the* maximum frequency *'o'*).

Reading out

Command 'Zu' is used to read out the current valid value.

2.6.10 Setting the maximum frequency

Parameter

Character	Permissible values	Writable	Data type	Default value
'o'	1 to 1000000	Yes	u32 (integer)	1

Firmware response

Confirms the command through an echo.

Description

Specifies the maximum speed in Hertz (steps per second).

The maximum speed is reached after first passing through the acceleration ramp.

Supports higher frequencies in open-loop operation:

- 1/2 step: 32,000 Hz
- 1/4 step: 64,000 Hz
- 1/8 step: 128,000 Hz
- 1/16 step: 256,000 Hz
- 1/32 step: 512,000 Hz
- 1/64 step: 1,000,000 Hz

Reading out

Command 'Zo' is used to read out the current valid value.

2.6.11 Setting the maximum frequency 2

Parameter

Character	Permissible values	Writable	Data type	Default value
'n'	1 to 1000000	Yes	u32 (integer)	1

Firmware response

Confirms the command through an echo.

Description

Specifies the maximum speed 2 in Hertz (steps per second).

The maximum speed 2 is reached after first passing through the acceleration ramp.

Supports higher frequencies in open-loop operation:

- 1/2 step: 32,000 Hz
- 1/4 step: 64,000 Hz
- 1/8 step: 128,000 Hz
- 1/16 step: 256,000 Hz
- 1/32 step: 512,000 Hz
- 1/64 step: 1,000,000 Hz

This value is only applied in the flag positioning mode. See command *2.6.6 Setting positioning* mode *'p'*.

Reading out

Command 'Zn' is used to read out the current valid value.

2.6.12 Setting the acceleration ramp

Parameters

Character	Permissible values	Writable	Data type	Default value
'b'	1 to 65535	Yes	u16 (integer)	1

Firmware response

Confirms the command through an echo.

Description

Specifies the acceleration ramp.

To convert the parameter to acceleration in Hz/ms, the following formula is used:

Acceleration in Hz/ms = ((3000.0 / sqrt((float)<parameter>)) - 11.7).

Reading out

Command 'Zb' is used to read out the current valid value.

2.6.13 Setting the brake ramp

Parameters

Character	Permissible values	Writable	Data type	Default value
'B'	0 to 65535	Yes	u16 (integer)	0

Firmware response

Confirms the command through an echo.

Description

Specifies the brake ramp.

Reading out

Command 'ZB' is used to read out the current valid value.

2.6.14 Setting the quickstop ramp

Parameters

Character	Permissible values	Writable	Data type	Default value
'H'	0 to 8000	Yes	u16 (integer)	0

Firmware response

Confirms the command through an echo.

Description

Specifies the quickstop ramp.

Quickstop: Used, for example, if the limit switch is overrun.

Reading out

Command 'ZH' is used to read out the current valid value.

2.6.15 Setting the direction of rotation

Parameter

Character	Permissible values	Writable	Data type	Default value
'd'	0 and 1	Yes	u8 (integer)	0

Firmware response

Confirms the command through an echo.

Description

Sets the direction of rotation:

0: Left

1: Right

Reading out

Command 'Zd' is used to read out the current valid value.

2.6.16 Setting the change of direction

Parameter

Character	Permissible values	Writable	Data type	Default value
't'	0 and 1	Yes	u8 (integer)	0

Firmware response

Confirms the command through an echo.

Description

With repetition records, the rotation direction of the motor is reversed with every repetition if this parameter is set to '1'. See command *2.6.17 Setting the* repetitions *'W'*.

Reading out

Command 'Zt' is used to read out the current valid value.

2.6.17 Setting the repetitions

Parameter

Character	Permissible values	Writable	Data type	Default value
'W'	0 to 254	Yes	u8 (integer)	0

Firmware response

Confirms the command through an echo.

Description

Specifies the number of repetitions of the current record.

A value of 0 indicates an endless number of repetitions.

Normally, the value is set to 1 for one repetition.

Reading out

Command 'ZW' is used to read out the current valid value.

2.6.18 Setting the record pause

Parameter

Character	Permissible values	Writable	Data type	Default value
'P'	0 to 65535	Yes	u16 (integer)	0

Firmware response

Confirms the command through an echo.

Description

Specifies the pause between record repetitions or between a record and a continuation record in ms (milliseconds).

If a record does not have a continuation record or a repetition, the pause is not executed and the motor is ready again immediately after the end of the run.

Reading out

Command 'ZP' is used to read out the current valid value.

2.6.19 Setting the continuation record

Parameter

Character	Permissible values	Writable	Data type	Default value
'N'	0 to 32	Yes	u8 (integer)	0

Firmware response

Confirms the command through an echo.

Description

Specifies the number of the continuation record. If the parameter is set to '0', a continuation record is not performed.

Reading out

Command 'ZN' is used to read out the current valid value.

2.7 Mode-specific commands

2.7.1 Setting the dead range for the joystick mode

Parameter

Character	Permissible values	Writable	Data type	Default value
'='	0 to 100	Yes	u8 (integer)	0

Firmware response

Confirms the command through an echo.

Description

Sets the dead range in joystick mode.

In joystick mode, the motor can be moved forward and backward via a voltage on the analog input.

The value range halfway between the maximum and minimum voltages in which the motor does not rotate is the dead range. It is specified as a percentage of the range width.

Reading out

Command 'Z=' is used to read out the current setting of the dead range.

2.7.2 Setting the filter for the analog and joystick modes

Parameter

Character	Permissible values	Writable	Data type	Default value
'f'	0 to 255	Yes	u8 (integer)	0

Firmware response

Confirms the command through an echo.

Description

In the analog and joystick modes, the analog input is used to set the speed. Command 'f' is used to set the number of samples averaged to determine the final value.

Reading out

Command 'Zf' is used to read out the current setting of the value.

f = 8-bit (Bit 0-3: number of samples; Bit 4-7: size of the hysteresis) + 16

Example: f=50: Smoothing: Recursive filter over 4 values
f=84: Strong smoothing: Recursive filter over 16 values

2.7.3 Setting the minimum voltage for the analog mode

Parameter

Character	Permissible values	Writable	Data type	Default value
'Q'	-100 to +100	Yes	s8 (integer)	-100

Firmware response

Confirms the command through an echo.

Description

Specifies the beginning of the range of the analog input in 0.1V steps.

Reading out

Command 'ZQ' is used to read out the current valid value.

2.7.4 Setting the maximum voltage for the analog mode

Parameter

Character	Permissible values	Writable	Data type	Default value
'R'	-100 to +100	Yes	s8 (integer)	100

Firmware response

Confirms the command through an echo.

Description

Specifies the end of the range of the analog input in 0.1V steps.

Reading out

Command 'ZR' is used to read out the current valid value.

2.7.5 Resetting switch-on numerator

Parameters

Character	Permissible values	Writable	Data type	Default value
'%'	1	Yes	u32 (integer)	1

Firmware response

Confirms the command through an echo.

Description

The switch-on numerator is incremented by “1” each time the current is switched on and specifies how often the controller has been switched on since the last reset.
If the value is set to '1', the switch-on numerator is reset to “0”.

Reading out

Command 'Z%' is used to read out the current valid value.

2.7.6 Adjusting the time until the current reduction

Parameters

Character	Permissible values	Writable	Data type	Default value
'G'	0 to 10000	Yes	u16 (integer)	80

Unit

ms

Firmware response

Confirms the command through an echo.

Description

The value defines the wait time at standstill until the current is reduced.

Reading out

Command 'ZG' is used to read out the current valid value.

2.7.7 Increasing the speed

Parameter

Character	Permissible values	Writable	Data type	Default value
'+'	–	No	–	–

Firmware response

Confirms the command through an echo.

Description

Increases the speed in the speed mode by 100 steps/s.

2.7.8 Reducing the speed

Parameter

Character	Permissible values	Writable	Data type	Default value
'-'	–	No	–	–

Firmware response

Confirms the command through an echo.

Description

Decreases the speed in the speed mode by 100 steps/s.

2.7.9 Actuating the trigger

Parameter

Character	Permissible values	Writable	Data type	Default value
'T'	–	No	–	–

Firmware response

Confirms the command through an echo.

Description

Trigger for the flag positioning mode.

Before triggering, the motor travels at a constant speed.

After triggering, the motor finishes travelling the set distance from the position where triggering occurred, and then stops.

2.8 Commands for JAVA program

2.8.1 Transferring a Java program to the controller

Parameters

Character	Permissible values	Writable	Data type	Default value
'(J'	0 to 268500991	Yes	s32 (integer)	0

Firmware response

Confirms the command through an echo.

Description

Carried out independently by NanoPro or NanoJEasy.

2.8.2 Starting a loaded Java program

Parameters

Character	Permissible values	Writable	Data type	Default value
'(JA'	0	No	u8 (integer)	0

Firmware response

Confirms the command with “(JA+” if the program was successfully started or with “(JA-” if the program could not be started (no valid program or no program at all loaded in the controller).

Description

The command starts the Java program loaded in the controller.

2.8.3 Stopping the running Java program

Parameters

Character	Permissible values	Writable	Data type	Default value
'(JS'	0	No	u8 (integer)	0

Firmware response

Confirms the command with “(JS+” if the program was successfully stopped or with “(JS-” if the program had already terminated.

Description

The command stops the Java program that is currently running.

2.8.4 Verifying loaded Java program

Parameters

Character	Permissible values	Writable	Data type	Default value
'(JI'	0	No	u8 (integer)	0

Firmware response

In response to the command, the controller returns “ECAFFE01” as the program ID.

Description

The command loads the current program from the EEPROM and initializes the VM. This initialization is also carried out automatically when switching on the controller and when transferring the program to the PD4 utility.

2.8.5 Automatically starting the Java program when switching on the controller

Parameters

Character	Permissible values	Writable	Data type	Default value
'(JB'	0 to 255	Yes	u8 (integer)	0

Firmware response

Confirms the command with “(JB=1” if the program is started automatically, or with “(JB=0” if the program is not started automatically.

Description

This command is used to specify whether the program is to be started automatically:

- “0” = do not start program automatically
- “1” = start program automatically

2.8.6 Reading out error of the Java program

Parameters

Character	Permissible values	Writable	Data type	Default value
'(JE'	0 to 255	No	u8 (integer)	0

Firmware response

Returns the index of the error memory with the last error that occurred. See Section *3.8 Possible Java* error messages.

Description

This command reads out the last error.

2.8.7 Reading out the warning of the Java program

Parameters

Character	Permissible values	Writable	Data type	Default value
'(JW'	0 to 255	No	u8 (integer)	0

Firmware response

Returns the last warning that occurred. Currently only:

- “0” = no warning
- “WARNING_FUNCTION_NOT_SUPPORTED”

Description

This command reads out the last warning.

2.9 Closed loop settings

2.9.1 Activating closed-loop mode

Parameters

Character	Permissible values	Writable	Data type	Default value
':CL_enable'	0 and 1	Yes	u8 (integer)	0

Firmware response

Confirms the command through an echo.

Description

If the value is set to "1", the firmware is instructed to activate the closed loop. This is only possible if a special reference run was performed since the unit was last switched on (mode 8 "!8").

Important conditions

The following conditions must be met when activating the closed loop:

- The "CL_Motor_pp", "CL_rotenc_inc" and "CL_rotenc_rev" settings must agree with the technical data of the connected stepper motor.
  For more information, see commands *2.9.9 Setting the motor pole* pairs, *2.9.10 Setting the number of* increments and *2.9.11 Setting the number of* revolutions.
- Every time a new motor is connected (even if it is the same type), a calibration run must be performed (mode 101 "!101").

ATTENTION:
If one of these conditions is not met, the motor may accelerate to a level that exceeds its maximum mechanical load capacity!

Reading out

If the keyword is sent without a “= + value”, the current setting of the value can be read out.

2.9.2 Reading out the closed loop mode status

Parameters

Character	Permissible values	Writable	Data type	Default value
':CL_is_enabled'	0 and 1	No	u8 (integer)	0

Firmware response

Returns the status:

- “0” = not enabled
- “1” = enabled

Description

Reads out the status of the closed loop mode.

2.9.3 Setting the tolerance window for the limit position

Parameters

Character	Permissible values	Writable	Data type	Default value
':CL_position_window'	0 to 2147483647	Yes	u32 (integer)	0

Unit

Increments

Firmware response

Confirms the command through an echo.

Description

If the closed loop is active, this is a criterion for when the firmware considers the limit position to have been reached. The parameter defines a tolerance window in increments of the encoder.

If the position actually measured is within the desired limit position + – the tolerance that is set in this parameter, and if this condition is met over a certain period, the limit position is considered to have been reached.

The time for this time window is set in the "CL_position_window_time" parameter. See command *2.9.4 Setting the time for the tolerance* window of the limit position.

Reading out

If the keyword is sent without a “= + value”, the current setting of the value can be read out.

2.9.4 Setting the time for the tolerance window of the limit position

Parameters

Character	Permissible values	Writable	Data type	Default value
':CL_position_window_time'	0 to 65535	Yes	u16 (integer)	0

Unit

ms

Firmware response

Confirms the command through an echo.

Description

Specifies the time in milliseconds for the "CL_position_window" parameter.
For more information, see command *2.9.3 Setting the tolerance window* for the limit position.

Reading out

If the keyword is sent without a “= + value”, the current setting of the value can be read out.

2.9.5 Setting the maximum allowed following error

Parameters

Character	Permissible values	Writable	Data type	Default value
':CL_following_error_window'	0 to 2147483647	Yes	u32 (integer)	100

Unit

Increments

Firmware response

Confirms the command through an echo.

Description

If the closed loop is active, this parameter defines the maximum allowed following error in increments of the encoder.

If, at a certain point in time, the actual position differs from the setpoint position by more than this parameter, a position error is output and the closed loop is switched off.

In addition, the "CL_following_error_timeout" parameter can be used to specify for how long the following error may be larger than the tolerance without triggering a position error. See command *2.9.6 Setting the time for the* maximum following error.

Reading out

If the keyword is sent without a “= + value”, the current setting of the value can be read out.

2.9.6 Setting the time for the maximum following error

Parameters

Character	Permissible values	Writable	Data type	Default value
':CL_following_error_timeout'	0 to 65535	Yes	u16 (integer)	100

Unit

ms

Firmware response

Confirms the command through an echo.

Description

This parameter can be used to specify in milliseconds for how long the following error may be greater than the tolerance without triggering a position error. See command *2.9.5 Setting the maximum allowed* following error.

Reading out

If the keyword is sent without a "= + value", the current setting of the value can be read out.

2.9.7 Maximum speed deviation

Parameters

Character	Permissible values	Writable	Data type	Default value
':CL_speed_error_window'	0 to 2147483647	Yes	u32 (integer)	150

Unit

Increments

Firmware response

Confirms the command through an echo.

Description

If the closed loop is active, this parameter defines the maximum allowed speed deviation.

In addition, the ":CL_speed_error_timeout" can be used to specify for how long the speed deviation may be greater than the tolerance. For more information, see command *2.9.8 Time for maximum speed deviation*.

Reading out

If the keyword is sent without a "= + value", the current setting of the value can be read out.

2.9.8 Time for maximum speed deviation

Parameters

Character	Permissible values	Writable	Data type	Default value
':CL_speed_error_timeout'	0 to 65535	Yes	u16 (integer)	250

Unit

ms

Firmware response

Confirms the command through an echo.

Description

This parameter can be used to specify in milliseconds for how long the speed deviation may be greater than the tolerance. For more information, see command *2.9.7 Maximum speed deviation*.

Reading out

If the keyword is sent without a “= + value”, the current setting of the value can be read out.

2.9.9 Setting the motor pole pairs

Parameters

Character	Permissible values	Writable	Data type	Default value
':CL_motor_pp'	1 to 65535	Yes	u16 (integer)	50

Unit

Number of pole pairs

Firmware response

Confirms the command through an echo.

Description

The parameter sets the number of pole pairs of the connected motor.

Note:
After this parameter is changed, the firmware **must** be restarted (disconnect power).

The number of pole pairs equals 1/4 of the number of full steps per revolution. The adjustable values are currently 50 and 100. If other values are set, this will result in the closed loop not functioning properly. However, even in this case, a conversion for the error correction without the closed loop will still function.

This parameter corresponds with the command *2.5.8 Setting the* step angle 'a'.
If the "CL_motor_pp" or 'a' parameter is changed, the associated parameter is also changed.

The values are converted according to the following formula:

$$\frac{\text{CL\_motor\_pp} = 900}{\text{COMM\_CMD\_SETSTEPANGLE}}$$

Reading out

If the keyword is sent without a “= + value”, the current setting of the value can be read out.

2.9.10 Setting the number of increments

Parameters

Character	Permissible values	Writable	Data type	Default value
':CL_rotenc_inc'	1 to 65535	Yes	u16 (integer)	2000

Unit

Increments

Firmware response

Confirms the command through an echo.

Description

This parameter specifies the number of increments of the encoder for a specific number of revolutions. The number of revolutions can be set using the "CL_rotenc_rev" parameter. See command *2.9.11 Setting the number of* revolutions.

Currently, the values 1600 and 2000 are supported for the closed loop. If other values are set, this will result in the closed loop not functioning properly. However, even in this case, a conversion for the error correction without the closed loop will still function.

Note:
After this parameter is changed, the firmware **must** be restarted (disconnect power).

Reading out

If the keyword is sent without a “= + value”, the current setting of the value can be read out.

2.9.11 Setting the number of revolutions

Parameters

Character	Permissible values	Writable	Data type	Default value
':CL_rotenc_rev'	1	Yes	u16 (integer)	1

Unit

Revolutions

Firmware response

Confirms the command through an echo.

Description

This parameter specifies the number of revolutions for the "CL_rotenc_inc" parameter. See command *2.9.10 Setting the number of* increments.

This setting is available for compatibility reasons. It should always be set to "1". If other values are set, this will result in the closed loop not functioning properly. However, even in this case, a conversion for the error correction without the closed loop will still function.

Note:
After this parameter is changed, the firmware **must** be restarted (disconnect power).

Reading out

If the keyword is sent without a “= + value”, the current setting of the value can be read out.

2.9.12 Setting the numerator of the P component of the speed controller

Parameters

Character	Permissible values	Writable	Data type	Default value
':CL_KP_v_Z'	0 to 65535	Yes	u16 (integer)	1

Unit

Numerator

Firmware response

Confirms the command through an echo.

Description

This parameter specifies the numerator of the proportional component of the speed controller.

Reading out

If the keyword is sent without a “= + value”, the current setting of the value can be read out.

2.9.13 Setting the denominator of the P component of the speed controller

Parameters

Character	Permissible values	Writable	Data type	Default value
':CL_KP_v_N'	0 to 15	Yes	u8 (integer)	3

Unit

Denominator as a power of 2

Firmware response

Confirms the command through an echo.

Description

This parameter specifies the denominator of the proportional component of the speed controller as a power of 2.

0 = 1

1 = 2

2 = 4

3 = 8

etc.

Reading out

If the keyword is sent without a “= + value”, the current setting of the value can be read out.

2.9.14 Setting the numerator of the I component of the speed controller

Parameters

Character	Permissible values	Writable	Data type	Default value
':CL_KI_v_Z'	0 to 65535	Yes	u16 (integer)	1

Unit

Numerator

Firmware response

Confirms the command through an echo.

Description

This parameter specifies the numerator of the integral component of the speed controller.

Reading out

If the keyword is sent without a “= + value”, the current setting of the value can be read out.

2.9.15 Setting the denominator of the I component of the speed controller

Parameters

Character	Permissible values	Writable	Data type	Default value
':CL_KI_v_N'	0 to 15	Yes	u8 (integer)	4

Unit

Denominator as a power of 2

Firmware response

Confirms the command through an echo.

Description

This parameter specifies the denominator of the integral component of the speed controller as a power of 2.

0 = 1

1 = 2

2 = 4

3 = 8

etc.

Reading out

If the keyword is sent without a “= + value”, the current setting of the value can be read out.

2.9.16 Setting the numerator of the D component of the speed controller

Parameters

Character	Permissible values	Writable	Data type	Default value
':CL_KD_v_Z'	0 to 65535	Yes	u16 (integer)	0

Unit

Numerator

Firmware response

Confirms the command through an echo.

Description

This parameter specifies the numerator of the differential component of the speed controller.

Reading out

If the keyword is sent without a “= + value”, the current setting of the value can be read out.

2.9.17 Setting the denominator of the D component of the speed controller

Parameters

Character	Permissible values	Writable	Data type	Default value
':CL_KD_v_Z'	0 to 15	Yes	u8 (integer)	0

Unit

Denominator as a power of 2

Firmware response

Confirms the command through an echo.

Description

This parameter specifies the denominator of the differential component of the speed controller as a power of 2.

0 = 1

1 = 2

2 = 4

3 = 8

etc.

Reading out

If the keyword is sent without a “= + value”, the current setting of the value can be read out.

2.9.18 Setting the numerator of the P component of the cascading speed controller

Parameters

Character	Permissible values	Writable	Data type	Default value
':CL_KP_csv_Z'	0 to 65535	Yes	u16 (integer)	0

Unit

Numerator

Firmware response

Confirms the command through an echo.

Description

This parameter specifies the numerator of the proportional component of the cascading speed controller.

Reading out

If the keyword is sent without a “= + value”, the current setting of the value can be read out.

2.9.19 Setting the denominator of the P component of the cascading speed controller

Parameters

Character	Permissible values	Writable	Data type	Default value
':CL_KP_csv_N'	0 to 15	Yes	u8 (integer)	0

Unit

Denominator as a power of 2

Firmware response

Confirms the command through an echo.

Description

This parameter specifies the denominator of the proportional component of the cascading speed controller as a power of 2.

0 = 1

1 = 2

2 = 4

3 = 8

etc.

Reading out

If the keyword is sent without a “= + value”, the current setting of the value can be read out.

2.9.20 Setting the numerator of the I component of the cascading speed controller

Parameters

Character	Permissible values	Writable	Data type	Default value
':CL_KI_csv_Z'	0 to 65535	Yes	u16 (integer)	0

Unit

Numerator

Firmware response

Confirms the command through an echo.

Description

This parameter specifies the numerator of the integral component of the cascading speed controller.

Reading out

If the keyword is sent without a “= + value”, the current setting of the value can be read out.

2.9.21 Setting the denominator of the I component of the cascading speed controller

Parameters

Character	Permissible values	Writable	Data type	Default value
':CL_KI_csv_N'	0 to 15	Yes	u8 (integer)	0

Unit

Denominator as a power of 2

Firmware response

Confirms the command through an echo.

Description

This parameter specifies the denominator of the integral component of the cascading speed controller as a power of 2.

0 = 1

1 = 2

2 = 4

3 = 8

etc.

Reading out

If the keyword is sent without a “= + value”, the current setting of the value can be read out.

2.9.22 Setting the numerator of the D component of the cascading speed controller

Parameters

Character	Permissible values	Writable	Data type	Default value
':CL_KD_csv_Z'	0 to 65535	Yes	u16 (integer)	0

Unit

Numerator

Firmware response

Confirms the command through an echo.

Description

This parameter specifies the numerator of the differential component of the cascading speed controller.

Reading out

If the keyword is sent without a "= + value", the current setting of the value can be read out.

2.9.23 Setting the denominator of the D component of the cascading speed controller

Parameters

Character	Permissible values	Writable	Data type	Default value
':CL_KD_csv_N'	0 to 15	Yes	u8 (integer)	0

Unit

Denominator as a power of 2

Firmware response

Confirms the command through an echo.

Description

This parameter specifies the denominator of the differential component of the cascading speed controller as a power of 2.

0 = 1

1 = 2

2 = 4

3 = 8

etc.

Reading out

If the keyword is sent without a "= + value", the current setting of the value can be read out.

2.9.24 Setting the numerator of the P component of the position controller

Parameters

Character	Permissible values	Writable	Data type	Default value
':CL_KP_s_Z'	0 to 65535	Yes	u16 (integer)	100

Unit

Numerator

Firmware response

Confirms the command through an echo.

Description

This parameter specifies the numerator of the proportional component of the position controller.

Reading out

If the keyword is sent without a “= + value”, the current setting of the value can be read out.

2.9.25 Setting the denominator of the P component of the position controller

Parameters

Character	Permissible values	Writable	Data type	Default value
':CL_KP_s_N'	0 to 15	Yes	u8 (integer)	0

Unit

Denominator as a power of 2

Firmware response

Confirms the command through an echo.

Description

This parameter specifies the denominator of the proportional component of the position controller as a power of 2.

0 = 1

1 = 2

2 = 4

3 = 8

etc.

Reading out

If the keyword is sent without a “= + value”, the current setting of the value can be read out.

2.9.26 Setting the numerator of the I component of the position controller

Parameters

Character	Permissible values	Writable	Data type	Default value
':CL_KI_s_Z'	0 to 65535	Yes	u16 (integer)	1

Unit

Numerator

Firmware response

Confirms the command through an echo.

Description

This parameter specifies the numerator of the integral component of the position controller.

Reading out

If the keyword is sent without a “= + value”, the current setting of the value can be read out.

2.9.27 Setting the denominator of the I component of the position controller

Parameters

Character	Permissible values	Writable	Data type	Default value
':CL_KI_s_N'	0 to 15	Yes	u8 (integer)	0

Unit

Denominator as a power of 2

Firmware response

Confirms the command through an echo.

Description

This parameter specifies the denominator of the integral component of the position controller as a power of 2.

0 = 1

1 = 2

2 = 4

3 = 8

etc.

Reading out

If the keyword is sent without a “= + value”, the current setting of the value can be read out.

2.9.28 Setting the numerator of the D component of the position controller

Parameters

Character	Permissible values	Writable	Data type	Default value
':CL_KD_s_Z'	0 to 65535	Yes	u16 (integer)	200

Unit

Numerator

Firmware response

Confirms the command through an echo.

Description

This parameter specifies the numerator of the differential component of the position controller.

Reading out

If the keyword is sent without a “= + value”, the current setting of the value can be read out.

2.9.29 Setting the denominator of the D component of the position controller

Parameters

Character	Permissible values	Writable	Data type	Default value
':CL_KD_s_N'	0 to 15	Yes	u8 (integer)	0

Unit

Denominator as a power of 2

Firmware response

Confirms the command through an echo.

Description

This parameter specifies the denominator of the differential component of the position controller as a power of 2.

0 = 1

1 = 2

2 = 4

3 = 8

etc.

Reading out

If the keyword is sent without a “= + value”, the current setting of the value can be read out.

2.9.30 Setting the numerator of the P component of the cascading position controller

Parameters

Character	Permissible values	Writable	Data type	Default value
':CL_KP_css_Z'	0 to 65535	Yes	u16 (integer)	0

Unit

Numerator

Firmware response

Confirms the command through an echo.

Description

This parameter specifies the numerator of the proportional component of the cascading position controller.

Reading out

If the keyword is sent without a “= + value”, the current setting of the value can be read out.

2.9.31 Setting the denominator of the P component of the cascading position controller

Parameters

Character	Permissible values	Writable	Data type	Default value
':CL_KP_css_N'	0 to 15	Yes	u8 (integer)	0

Unit

Denominator as a power of 2

Firmware response

Confirms the command through an echo.

Description

This parameter specifies the denominator of the proportional component of the cascading position controller as a power of 2.

0 = 1

1 = 2

2 = 4

3 = 8

etc.

Reading out

If the keyword is sent without a “= + value”, the current setting of the value can be read out.

2.9.32 Setting the numerator of the I component of the cascading position controller

Parameters

Character	Permissible values	Writable	Data type	Default value
':CL_KI_css_Z'	0 to 65535	Yes	u16 (integer)	0

Unit

Numerator

Firmware response

Confirms the command through an echo.

Description

This parameter specifies the numerator of the integral component of the cascading position controller.

Reading out

If the keyword is sent without a “= + value”, the current setting of the value can be read out.

2.9.33 Setting the denominator of the I component of the cascading position controller

Parameters

Character	Permissible values	Writable	Data type	Default value
':CL_KI_css_N'	0 to 15	Yes	u8 (integer)	0

Unit

Denominator as a power of 2

Firmware response

Confirms the command through an echo.

Description

This parameter specifies the denominator of the integral component of the cascading position controller as a power of 2.

0 = 1

1 = 2

2 = 4

3 = 8

etc.

Reading out

If the keyword is sent without a “= + value”, the current setting of the value can be read out.

2.9.34 Setting the numerator of the D component of the cascading position controller

Parameters

Character	Permissible values	Writable	Data type	Default value
':CL_KD_css_Z'	0 to 65535	Yes	u16 (integer)	0

Unit

Numerator

Firmware response

Confirms the command through an echo.

Description

This parameter specifies the numerator of the differential component of the cascading position controller.

Reading out

If the keyword is sent without a “= + value”, the current setting of the value can be read out.

2.9.35 Setting the denominator of the D component of the cascading position controller

Parameters

Character	Permissible values	Writable	Data type	Default value
':CL_KD_css_N'	0 to 15	Yes	u8 (integer)	0

Unit

Denominator as a power of 2

Firmware response

Confirms the command through an echo.

Description

This parameter specifies the denominator of the differential component of the cascading position controller as a power of 2.

0 = 1

1 = 2

2 = 4

3 = 8

etc.

Reading out

If the keyword is sent without a “= + value”, the current setting of the value can be read out.

2.10 Motor-dependent correction values determined by test runs for the closed loop mode

General information

The first time a controller with the associated motor is used, a test run must be started. Here, motor-dependent correction values are determined by the controller and stored.

These correction values can be read and stored with NanoPro in order to be able to write them back again if the controller is changed.

2.10.1 Reading out the encoder/motor offset

Parameters

Character	Permissible values	Writable	Data type	Default value
':CL_poscnt_offset'	-32768 to +32767	Yes	s16 (integer)	0

Firmware response

Confirms the command through an echo.

Description

The offset between the encoder and motor determined during the test run is read out.

2.10.2 Reading out the load angle of the motor

Parameters

Character	Permissible values	Writable	Data type	Default value
':CL_la_a' to **':CL_la_j'**	-32768 to +32767	Yes	s16 (integer)	0

Firmware response

Confirms the command through an echo.

Description

The speed-dependent data of the load angle of the motor (closed-loop load angle) determined during the test run are read out:

- CL_la_a
- CL-la_b
- CL-la_c
- CL-la_d
- CL-la_e
- CL-la_f
- CL-la_g
- CL-la_h
- CL-la_i
- CL-la_j

2.10.3 Reading out the correction values of the speed controller

Parameters

Character	Permissible values	Writable	Data type	Default value
':CL_ola_v_a' to **':CL_ola_v_g'**	-32768 to +32767	Yes	s16 (integer)	0

Firmware response

Confirms the command through an echo.

Description

The data of the load angle of the speed controller (closed-loop load angle velocity) determined during the test run are read out:

- CL_ola_v_a
- CL_ola_v_b
- CL_ola_v_c
- CL_ola_v_d
- CL_ola_v_e
- CL_ola_v_f
- CL_ola_v_g

2.10.4 Reading out the correction values of the current controller

Parameters

Character	Permissible values	Writable	Data type	Default value
':CL_ola_i_a' to **':CL_ola_i_g'**	-32768 to +32767	Yes	s16 (integer)	0

Firmware response

Confirms the command through an echo.

Description

The data of the load angle of the current controller (closed-loop load angle current) determined during the test run are read out:

- CL_ola_i_a
- CL_ola_i_b
- CL_ola_i_c
- CL_ola_i_d
- CL_ola_i_e
- CL_ola_i_f
- CL_ola_i_g

2.10.5 Reading out the correction values of the position controller

Parameters

Character	Permissible values	Writable	Data type	Default value
':CL_ola_l_a' to **':CL_ola_l_g'**	-2147483648 to +2147483647	Yes	s32 (integer)	0

Firmware response

Confirms the command through an echo.

Description

The data of the load angle of the position controller (closed-loop load angle position) determined during the test run are read out:

- CL_ola_l_a
- CL_ola_l_b
- CL_ola_l_c
- CL_ola_l_d
- CL_ola_l_e
- CL_ola_l_f
- CL_ola_l_g

2.11 Scope mode

2.11.1 Integration of a scope

Description

In the scope mode, the values to be measured are selected and transferred to the motor. The motor then carries out a measurement and returns the result in real time to the NanoPro controller software.

- The transferred data are binary.
- The data are transferred in the order of priority.
- The last data byte of each data packet contains a CRC8 checksum.

Examples

Each data source can be selected separately:

:Capt_Time=10 → sends the selected data every 10 ms.

:Capt_Time=0 → ends the scope mode

:Capt_sPos=1 → the setpoint position is selected

:Capt_sPos=0 → the setpoint position is deselected

By default no data source is selected.

Data word when :Capt_sCurr=1 and :Capt_iIn=1

:Capt_sCurr_BYTE

:Capt_iIn_BYTE_HI

:Capt_iIn_BYTE_LO CRC

2.11.2 Setting the sample rate

Parameters

Character	Permissible values	Writable	Data type	Default value
':Capt_Time'	0 to 65535	Yes	u16 (integer)	0

Priority

–

Unit

ms (milliseconds)

Description

The parameter defines the time interval in ms in which the selected data are sent. The value range is "Unsigned 16".

"0" deactivates the scope function.

Example

:Capt_Time=10 sends the selected data every 10 ms.

:Capt_Time=0 ends the scope mode

Reading out

If the keyword is sent without a “= + value”, the current setting of the value can be read out.

2.11.3 Reading out the setpoint position of the ramp generator

Parameters

Character	Permissible values	Writable	Data type	Default value
':Capt_sPos'	0 and 1	Yes	u8 (integer)	0

Priority

1

Unit

Steps

Description

Delivers the setpoint position generated by the ramp generator.

Example

:Capt_sPos=1 the setpoint position is selected

:Capt_sPos=0 the setpoint position is deselected

2.11.4 Reading out the actual position of the encoder

Parameters

Character	Permissible values	Writable	Data type	Default value
':Capt_iPos'	0 and 1	Yes	u8 (integer)	0

Priority

2

Unit

Steps

Description

Returns the current encoder position.

Example

:Capt_iPos=1 the actual position is selected

:Capt_iPos=0 the actual position is deselected

2.11.5 Reading out the setpoint current of the motor controller

Parameters

Character	Permissible values	Writable	Data type	Default value
':Capt_sCurr'	0 and 1	Yes	u8 (integer)	0

Priority

3

Unit

None

32767 corresponds to 150% of the maximum current (the value can also be negative).

Description

Delivers the setpoint current used for driving the motor.

Example

:Capt_sCurr=1 the setpoint current is selected

:Capt_sCurr=0 the setpoint current is deselected

2.11.6 Reading out the actual voltage of the controller

Parameters

Character	Permissible values	Writable	Data type	Default value
':Capt_iVolt'	0 and 1	Yes	u8 (integer)	0

Priority

4

Unit

Value range 0 – 1023 (10-bit)

1023 is equivalent to 66.33 V

0 is equivalent to 0 V

Description

Delivers the voltage applied at the controller.

Example

:Capt_iVolt=1 the applied voltage is selected

:Capt_iVolt=0 the applied voltage is deselected

2.11.7 Reading out the digital inputs

Parameters

Character	Permissible values	Writable	Data type	Default value
':Capt_iIn'	0 and 1	Yes	u8 (integer)	0

Priority

5

Unit

None

Description

Delivers the bit mask of the inputs.

Example

:Capt_iIn=1 the bit mask of the inputs is selected

:Capt_iIn=0 the bit mask of the inputs is deselected

2.11.8 Reading out the voltage at the analog input

Parameters

Character	Permissible values	Writable	Data type	Default value
':Capt_iAnalog'	0 and 1	Yes	u8 (integer)	0

Priority

6

Unit

0 is equivalent to 0 V

1023 is equivalent to +10 V

Description

Delivers the voltage of the analog input.

Example

:Capt_iAnalog=1 the voltage of the analog input is selected

:Capt_iAnalog=0 the voltage of the analog input is deselected

2.11.9 Reading out the CAN bus load

Parameters

Character	Permissible values	Writable	Data type	Default value
':Capt_iBus'	0 and 1	Yes	u8 (integer)	0

Priority

7

Unit

%

Invalid values are ignored.

Description

Delivers the approximate degree of utilisation of the CAN bus in %.

Example

:Capt_iBus=1 the utilisation of the CAN bus is selected

:Capt_iBus=0 the utilisation of the CAN bus is deselected

2.11.10 Reading out the controller temperature

Parameters

Character	Permissible values	Writable	Data type	Default value
':Capt_ITemp'	0 and 1	Yes	u8 (integer)	0

Priority

8

Unit

Value range 0 – 1023

295 = 75 °C

261 = 80 °C

Description

Delivers the temperature measured in the controller.

Example

:Capt_ITemp=1 the temperature of the controller is selected

:Capt_ITemp=0 the temperature of the controller is deselected

2.11.11 Reading out the following error

Parameters

Character	Permissible values	Writable	Data type	Default value
':Capt_IFollow'	0 and 1	Yes	u8 (integer)	0

Priority

9

Unit

Steps

Description

Delivers the difference between the setpoint and actual position.

Example

:Capt_IFollow=1 the difference between the setpoint and actual position is selected

:Capt_IFollow=0 the difference between the setpoint and actual position is deselected

2.12 Configuration of the current controller of the SMCP33 and PD4-N drivers

2.12.1 Setting the P component of the current controller at standstill

Parameters

Character	Permissible values	Writable	Data type	Default value
':dspdrive_KP_low'	0 to 65535	Yes	u16 (integer)	1

Firmware response

Confirms the command through an echo.

Description

This parameter can be used to set the P component of the current controller of the SMCP33 and PD-4N drivers at standstill.

Normally, no change necessary.

Reading out

If the keyword is sent without a “= + value”, the current setting of the value can be read out.

2.12.2 Setting the P component of the current controller during the run

Parameters

Character	Permissible values	Writable	Data type	Default value
':dspdrive_KP_hig'	0 to 65535	Yes	u16 (integer)	1

Firmware response

Confirms the command through an echo.

Description

This parameter can be used to set the P component of the current controller of the SMCP33 and PD-4N drivers during the run.

Normally, no change necessary.

Reading out

If the keyword is sent without a “= + value”, the current setting of the value can be read out.

2.12.3 Setting the scaling factor for speed-dependent adjustment of the P component of the controller during the run

Parameters

Character	Permissible values	Writable	Data type	Default value
':dspdrive_KP_scale'	0 to 65535	Yes	u16 (integer)	58

Firmware response

Confirms the command through an echo.

Description

This parameter can be used to set the scaling factor for the speed-dependent adjustment of the P component of the current controller of the SMCP33 and PD-4N drivers during the run.

Normally, no change necessary.

Reading out

If the keyword is sent without a “= + value”, the current setting of the value can be read out.

2.12.4 Setting the I component of the current controller at standstill

Parameters

Character	Permissible values	Writable	Data type	Default value
':dspdrive_KI_low'	0 to 65535	Yes	u16 (integer)	1

Firmware response

Confirms the command through an echo.

Description

This parameter can be used to set the I component of the current controller of the SMCP33 and PD-4N drivers at standstill.

Normally, no change necessary.

Reading out

If the keyword is sent without a “= + value”, the current setting of the value can be read out.

2.12.5 Setting the I component of the current controller during the run

Parameters

Character	Permissible values	Writable	Data type	Default value
':dspdrive_KI_hig'	0 to 65535	Yes	u16 (integer)	1

Firmware response

Confirms the command through an echo.

Description

This parameter can be used to set the I component of the current controller of the SMCP33 and PD-4N drivers during the run.

Normally, no change necessary.

Reading out

If the keyword is sent without a "= + value", the current setting of the value can be read out.

2.12.6 Setting the scaling factor for speed-dependent adjustment of the I component of the controller during the run

Parameters

Character	Permissible values	Writable	Data type	Default value
':dspdrive_KI_scale'	0 to 65535	Yes	u16 (integer)	200

Firmware response

Confirms the command through an echo.

Description

This parameter can be used to set the scaling factor for the speed-dependent adjustment of the I component of the current controller of the SMCP33 and PD-4N drivers during the run.

Normally, no change necessary.

Reading out

If the keyword is sent without a "= + value", the current setting of the value can be read out.

3 Programming with Java (NanoJEasy)

3.1 Overview

About this chapter

This chapter contains a brief overview of the programming language of the Nanotec stepper motor positioning controls.

The drivers contain a Java Virtual Machine (VM) that has been extended by some manufacturer-specific functions.

Restrictions

Due to the current level of development (Beta1) and the hardware that is used, the current VM is subject to the following restrictions:

- The program may have a maximum size of 4096 bytes after the linking.
- The stack and the heap are limited to 50 entries recursive function calls are only possible only to a limited extent.
- No threads are supported.

Abbreviations used

VM	Virtual Machine
Java SE	Java Standard Edition
JDK	Java Development Kit
JRE	Java Runtime Environment

Preconditions

In order to develop a program for the controller, the following preconditions must be fulfilled:

- NanoJEasy programming environment installed
- SMCI47-S
- SMCP33
- SMCI33

Simultaneous communication over the serial interface

NanoJ runs as a virtual machine irrespective of the actual firmware and communicates with this firmware via the same functions that are also called up from the serial interface.

A Java program can, therefore, run at the same time as the positioning control is receiving and processing serial commands.

Restrictions:

- Sending is not possible via Java.
- The same functions should not be used from the Java program and over the serial interface at the same time (e.g. changing step mode) since changes in the firmware require a certain amount of time and hence may cause undefined responses.

3.2 Command overview

A list of commands for programming with Java (NanoJEasy) can be found below:

3.3 Installing NanoJEasy

General information

NanoJEasy is a programming environment for the development of Java programs which can run on Nanotec stepper motor positioning controls and enable advanced programming of the drivers.

NanoJEasy includes the freely available Gnu-Java compiler (gcj) for the translation of Java programs.

Procedure

Carry out the installation as follows:

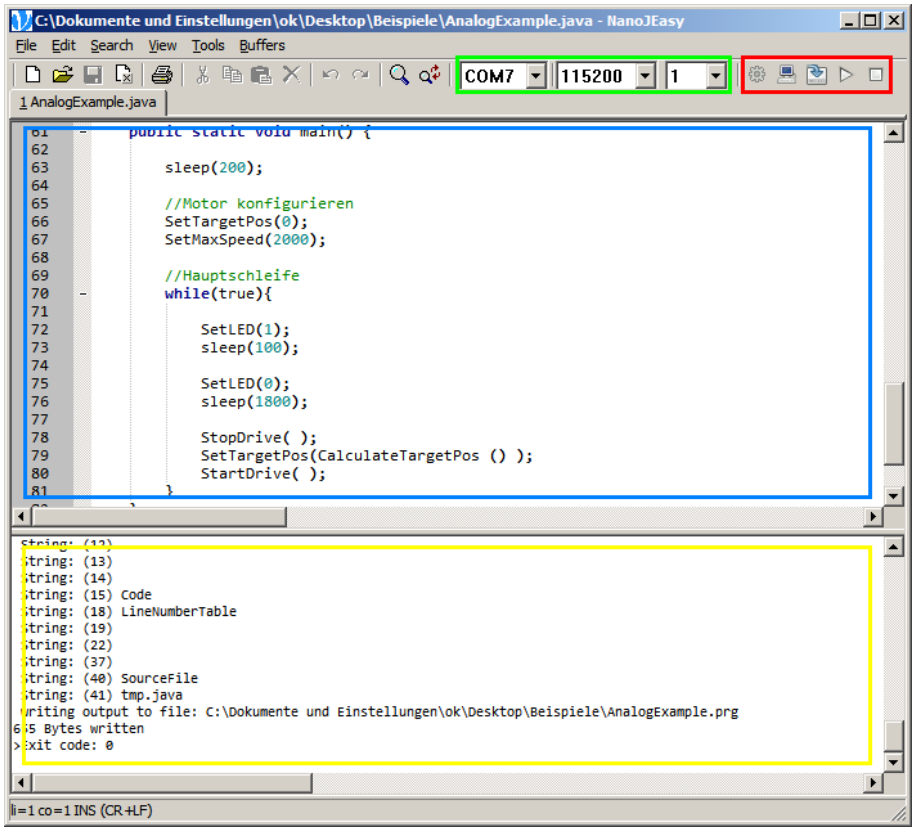
Step	Implementation
1	Double-click on the setup.exe file.
2	Select the desired language.
3	Confirm that you accept the license conditions.
4	Select the folder in which NanoJEasy should be installed.
5	Confirm or change the recommended start menu entry for NanoJEasy.
6	Start the installation.

3.4 Working with NanoJEasy

3.4.1 Main window of NanoJEasy

Screenshot

All important elements of the NanoJEasy main window are indicated in the following screenshot:



Explanation of the areas

- The following communication parameters can be set with the operating elements marked in green:
  - Selection of one of the existing COM ports
  - Selection of a baud rate
  - Selection of a motor number
- The following actions can be carried with the buttons marked in red:
  - Translation and linking of the current program
  - Simulation of the current program
  - Transfer of the current program into the controller
  - Execution of the program in the controller
  - Stoppage of the program running in the controller
- The program source text is edited in the text area marked in blue.
- Messages for the translation, simulation, transfer and execution of the developed program appear in the output area marked in yellow.

3.4.2 Development process with NanoJEasy

Development process

The development process with NanoJEasy normally follows the scheme shown below:

Level	Description
1	Create the program in the text area.
2	Translate and link the program.
3	Optional: Simulate the program.
4	Check the settings of the communication parameters.
5	Transfer the program to the controller.
6	Execute the program on the controller.

Important instructions for programming

The following instructions should always be observed during programming:

- Source text files must be created with the UTF-8 character encoding. NanoJEasy uses this character encoding as the default.
- The class name in the source text file must agree with the name of the source text file. Example: The “Testprogramm.java” file must contain the class “Test program class”.
- The Java commands for communication with the controller only initiate the respective action of the controller, but do not wait until the controller has carried out the action. If the Java program should wait until the action is carried out, a waiting time must be inserted after the command for execution, e.g. “Sleep(2000);". For more details, see also the example programs.

Completing the command on entry

Enter a command as follows:

Step	Implementation
1	Enter the first letters of a command, e.g. “Set” of “SetCurrent”.
2	Press the <Ctrl> + <space> keys. A selection list of commands that begin with “Set” appears.
3	Mark a command in the selection list using the “Up” and “Down” arrow keys.
4	Press the “Enter” key to select the command.

Starting and ending the simulation

Proceed as follows to start and end the simulation:

Step	Implementation
1	Click on the “Start simulation” button (see above). The outputs of the emulator appear consecutively in the output area.
2	Press the <Ctrl> + <Pause> keys to end the simulation.

3.4.3 Integrated commands

Classes and functions

The VM contains integrated functions that can be used in the program. The functions are grouped into a total of four different classes which can be integrated in the source code.

The following sections provide information on the individual classes and the functions they include.

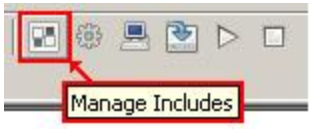
Integrating a class

The four different classes are included in the nanotec package and must be imported by the following entry at the start of the program:

```
import nanotec.*;
```

In addition, the classes which are really included on transfer to the controller must be selected in NanoJEasy.

“Manage Includes” button in the upper right area of the application



The “Manage Includes” opens.

The required classes can then be included simply by activating the checkbox:

Calling up functions

The individual functions of a class are called up in the source text as follows:

[Name of the class].[Name of the function]();

Example:

drive.StartDrive();

Integrating an individual function

To save memory space, the functions included in the classes can also be used individually.

To do so, every function that is to be used must be included as a declaration in the source code:

class example{

```
        //declaration of the function
        static native void StartDrive( );

        //main function
        //is called up at the start of the program
        public static void main() {

            //use of the function
            StartDrive( );
        }
}
```

3.5 Classes and functions

3.5.1 “comm” class

comm.SendInt

Declaration:

static native void SendInt(int in);

Sends the specified integer value over the serial interface.

comm.SendLong

Declaration:

static native void SendLong(long in);

Sends the specified long value over the serial interface.

3.5.2 “drive” class

drive.StartDrive

Declaration:

static native void StartDrive( );

This function starts the motor. The currently selected data record (mode, speed, ramp, etc.) is used here.

The function corresponds to the serial command 'A', see command *2.6.1 Starting the* motor.

drive.StopDrive

Declaration:

static native void StopDrive(int type);

Cancels the current travel; type determines how it will be stopped:

type = 0: A quickstop is carried out (braking with very steep ramp)

type = 1: Braking is carried out with the normal braking ramp

In the speed, analog and joystick modes, this is the only method of returning the motor to the ready state.

The motor is brought to an immediate halt without ramps. This may result in step loss at high speeds.

In the three modes named above the speed should, therefore, be reduced prior to the stop command.

The function corresponds to the serial command 'S', see command *2.6.2 Stopping a* motor.

drive.SetMaxSpeed

Declaration:

static native void SetMaxSpeed(int value);

Specifies the maximum speed in Hertz (steps per second).

The maximum speed is reached after first passing through the acceleration ramp.

The function corresponds to the serial command 'o<value>', see command *2.6.10 Setting the* maximum frequency.

drive.GetMaxSpeed

Declaration:

static native int GetMaxSpeed( );

Reads out the currently valid value of the maximum speed in Hertz (steps per second).

The function corresponds to the serial command 'Zo', see *2.3 Read* command.

drive.SetMinSpeed

Declaration:

static native void SetMinSpeed (int value);

Specifies the minimum speed in Hertz (steps per second) and can only be used in open loop mode.

At the start of a record the motor begins to turn with the minimum speed. It then accelerates up to the maximum speed with the set ramp.

The function corresponds to the serial command 'u<value>', see command *2.6.9 Setting the* minimum frequency.

drive.GetMinSpeed

Declaration:

static native int GetMinSpeed( );

Reads out the currently valid value of the minimum speed in Hertz (steps per second).

The function corresponds to the serial command 'Zu', see *2.3 Read* command.

drive.SetAccelaration

Declaration:

static native void SetAccelaration(int value);

Specifies the acceleration ramp (and at this time also the brake ramp).

To convert the parameter to acceleration in Hz/ms, the following formula is used:

Acceleration in Hz/ms = ((3000.0 / sqrt((float)<value>)) - 11.7).

The function corresponds to the serial command 'b<value>', see command *2.6.12 Setting the* acceleration ramp.

drive.GetAccelaration

Declaration:

static native int GetAccelaration( );

Reads out the currently valid value of the acceleration ramp.

The function corresponds to the serial command 'Zb', see *2.3 Read* command.

drive.SetTargetPos

Declaration:

static native void SetTargetPos(int value);

Specifies the travel distance in (micro)steps. Only positive values are allowed for the relative positioning. The direction is set with SetDirection.

For absolute positioning, this command specifies the target position. Negative values are allowed in this case. The direction of rotation set with SetDirection is ignored as this results from the current position and the target position.

The value range is from -100,000,000 to +100,000,000.

In the adaptive mode, this parameter refers to half steps.

The function corresponds to the serial command 's<value>', see command *2.6.8 Setting the* travel distance.

drive.GetTargetPos

Declaration:

static native int GetTargetPos( );

Reads out the currently valid value of the travel distance in (micro)steps.

The function corresponds to the serial command 'Zs', see *2.3 Read* command.

drive.SetDriveMode

Declaration:

static native void SetDriveMode(int value);

The function corresponds to the serial command '!<value>', see command *2.5.5 Setting the* motor mode.

Sets the motor mode. The following modes are available:

For old scheme:

1: Position mode

2: Speed mode

3: Flag positioning mode

4: Clock direction mode

5: Analog mode

6: Joystick mode

7: Analog positioning mode

8: HW reference mode

9: Torque mode

101: CL quick test mode

101: CL test mode

For more information, see command *2.6.6 Setting positioning mode (old* scheme) *'p'*.

For new scheme:

10: Motor mode

For more information, see command *2.6.7 Setting the positioning mode* (new scheme) *'p'*.

drive.GetDriveMode

Declaration:

static native int GetDriveMode( );

Reads out the current motor mode.

The function corresponds to the serial command 'Z!', see *2.3 Read* command.

drive.SetMode

Declaration:

static native void SetMode(int value);

The function corresponds to the serial command 'p<value>', see command *2.5.5 Setting the* motor mode.

The value combinations of the old scheme for motor mode '!' and positioning mode 'p' are:

Positioning mode (!=1)	
p=1	Relative positioning; The command *2.6.8 Setting the* travel distance 's' specifies the travel distance relative to the current position. The command *2.6.15 Setting the* direction of rotation 'd' specifies the direction. The parameter *2.6.8 Setting the* travel distance 's' must be positive.
p=2	Absolute positioning; Command *2.6.8 Setting the* travel distance *'s'* defines the target position relative to the reference position. Command *2.6.15 Setting the* direction of rotation *'d'* is ignored.
p=3	Internal reference run; The motor runs with the lower speed in the direction set in command *2.6.15 Setting the* direction of rotation *'d'* until it reaches the index line of the encoder. Then the motor runs a fixed number of steps to leave the index line again. For the direction of free travel, see command *2.5.6 Setting the* limit switch behavior *'l'*. This mode is only useful for motors with integrated and connected encoders.
p=4	External reference run; The motor runs at the highest speed in the direction set in command *2.6.15 Setting the* direction of rotation *'d'* until it reaches the limit switch. Then a free run is performed, depending on the setting. See command *2.5.6 Setting the* limit switch behavior *'l'*.
Speed mode (!=2)	
p=1	Speed mode; When the motor is started, the motor increases in speed to the maximum speed with the set ramp. Changes in the speed or direction of rotation are performed immediately with the set ramp without having to stop the motor first.
p=2	Not assigned
p=3	Internal reference run; see Positioning mode
p=4	External reference run; see Positioning mode

Flag positioning mode (!=3)	
p=1	Flag positioning mode; After starting, the motor runs up to the maximum speed. After arrival of the trigger event (command *2.7.9 Actuating the* trigger *'T'* or trigger input) the motor continues to travel the selected travel distance (command *2.6.8 Setting the* travel distance *'s'*) and changes its speed to the maximum speed 2 (command *2.6.11 Setting the maximum* frequency 2 *'n'*) for this purpose.
p=2	Not assigned
p=3	Internal reference run; see Positioning mode
p=4	External reference run; see Positioning mode
Clock direction mode (!=4)	
p=1	Manual left.
p=2	Manual right.
p=3	Internal reference run; see Positioning mode
p=4	External reference run; see Positioning mode
Analog mode (!=5)	
	Not applicable
Joystick mode (!=6)	
	Not applicable
Analog positioning mode (!=7)	
p=1	Analog positioning mode
p=2	Not assigned
p=3	Internal reference run; see Positioning mode
p=4	External reference run; see Positioning mode
HW reference mode (!=8)	
	Not applicable
Torque mode (!=9)	
	Not applicable
CL quick test mode (!=101)	
p=1	CL quick test mode
CL test mode (!=101)	
p=2	CL test mode

The value combinations of the new scheme for motor mode '!' and positioning mode 'p' are:

Positioning mode (!=10)	
p=1	Relative positioning; The command *2.6.8 Setting the* travel distance 's' specifies the travel distance relative to the current position. The command *2.6.15 Setting the* direction of rotation 'd' specifies the direction. The parameter *2.6.8 Setting the* travel distance 's' must be positive.
p=2	Absolute positioning; Command *2.6.8 Setting the* travel distance *'s'* defines the target position relative to the reference position. Command *2.6.15 Setting the* direction of rotation *'d'* is ignored.
p=3	Internal reference run; The motor runs with the lower speed in the direction set in command *2.6.15 Setting the* direction of rotation *'d'* until it reaches the index line of the encoder. Then the motor runs a fixed number of steps to leave the index line again. For the direction of free travel, see command *2.5.6 Setting the* limit switch behavior *'l'*. This mode is only useful for motors with integrated and connected encoders.
p=4	External reference run; The motor runs at the highest speed in the direction set in command *2.6.15 Setting the* direction of rotation *'d'* until it reaches the limit switch. Then a free run is performed, depending on the setting. See command *2.5.6 Setting the* limit switch behavior *'l'*.
Speed mode (!=10)	
p=5	Speed mode; When the motor is started, the motor increases in speed to the maximum speed with the set ramp. Changes in the speed or direction of rotation are performed immediately with the set ramp without having to stop the motor first.
p=3	Internal reference run; see Positioning mode
p=4	External reference run; see Positioning mode
Flag positioning mode (!=10)	
p=6	Flag positioning mode; After starting, the motor runs up to the maximum speed. After arrival of the trigger event (command *2.7.9 Actuating the* trigger *'T'* or trigger input) the motor continues to travel the selected travel distance (command *2.6.8 Setting the* travel distance *'s'*) and changes its speed to the maximum speed 2 (command *2.6.11 Setting the maximum* frequency 2 *'n'*) for this purpose.
p=3	Internal reference run; see Positioning mode
p=4	External reference run; see Positioning mode
Clock direction mode (!=10)	
p=7	Manual left.
p=8	Manual right.
p=9	Internal reference run; see Positioning mode

p=10	External reference run; see Positioning mode
Analog mode (!=10)	
p=11	Analog mode
Joystick mode (!=10)	
p=12	Joystick mode
Analog positioning mode (!=10)	
p=13	Analog positioning mode
p=3	Internal reference run; see Positioning mode
p=4	External reference run; see Positioning mode
HW reference mode (!=10)	
p=14	HW reference mode
Torque mode (!=10)	
p=15	Torque mode
CL quick test mode (!=10)	
p=16	CL quick test mode
CL test mode (!=10)	
p=17	CL test mode

drive.GetMode

Declaration:

static native int GetMode( );

Reads out the current positioning mode.

The function corresponds to the serial command 'Zp', see *2.3 Read* command.

drive.SetCurrent

Declaration:

static native void SetCurrent(int value);

Sets the phase current in percent. Values above 100 should be avoided.

The function corresponds to the serial command 'i<value>', see command *2.5.1 Setting the* phase current.

drive.GetCurrent

Declaration:

static native int GetCurrent( );

Reads out the currently selected phase current in percent.

The function corresponds to the serial command 'Zi', see *2.3 Read* command.

drive.SetCurrentReduction

Declaration:

static native void SetCurrentReduction(int value);

Sets the current of the current reduction at stanstill in percent. Like the phase current, this current is relative to the end value. Values above 100 should be avoided.

The function corresponds to the serial command 'r<value>', see command *2.5.2 Setting the phase current* at a standstill.

drive.GetCurrentReduction

Declaration:

static native int GetCurrentReduction( );

Reads out the currently selected phase current at standstill in percent.

The function corresponds to the serial command 'Zr', see *2.3 Read* command.

drive.GetStatus

Declaration:

static native int GetStatus( );

Returns the current status of the controller as a bit mask.

Bit 0 ready

Bit 1 reference

Bit 2 posError

Bit 3 endStartActive

Bit 4-7 mode

drive.SetDirection

Declaration:

static native void SetDirection(int value);

Sets the direction of rotation:

value=0 Direction of rotation, left

value=1 Direction of rotation, right

The function corresponds to the serial command 'd<value>', see command *2.6.15 Setting the* direction of rotation.

drive.GetDirection

Declaration:

static native int GetDirection( );

Reads out the currently set direction of rotation.

The function corresponds to the serial command 'Zd', see *2.3 Read* command.

drive.GetEncoderPosition

Declaration:

static native int GetEncoderPosition( );

Reads out the current position of the encoder.

The function corresponds to the serial command 'I', see command *2.5.16 Reading out* the encoder position.

drive.GetDemandPosition

Declaration:

static native int GetDemandPosition( );

Reads out the current position of the motor.

The function corresponds to the serial command 'C', see command *2.5.17 Reading out* the position.

drive.LoadDataSet

Declaration:

public static native void LoadDataSet (int whichone);

Parameter: int whichone 1-32

Return: None

Loads the selected data record into the controller. The data records can be configured by means of NanoPro.

3.5.3 "io" class

io.SetLED

Declaration:

static native void SetLED(int in);

Sets the error LED.

1: LED on

2: LED off

io.SetDigitalOutput

Declaration:

static native void SetDigitalOutput(int value);

Sets the digital outputs of the controller as bit-coded.

io.GetDigitalOutput

Declaration:

static native int GetDigitalOutput( );

Reads out the currently set bit mask for the digital outputs.

io.GetDigitalInput

Declaration:

static native int GetDigitalInput( );

Reads out the currently connected digital inputs.

io.GetAnalogInput

Declaration:

static native int GetAnalogInput(int Port);

Reads out the current values of the analog inputs. Port specifies the port to be read: 1 for the first analog port, 2 for the second port (if present).

3.5.4 "util" class

util.GetMillis

Declaration:

static native int GetMillis( );

Reads out the time since the controller was switched on in milliseconds.

util.Sleep

Declaration:

static void Sleep(int ms);

Waits for ms milliseconds.

util.TestBit

Declaration:

static boolean TestBit(int value, int whichone);

Checks that a bit is set.

value	=	value that the bit to be checked contains
whichone	=	specifies which bit should be tested 0 corresponds to the lowest bit
Return	=	true if the bit is set, otherwise false

util.SetBit

Declaration:

static int SetBit(int value, int whichone);

Sets a bit in an integer.

Value	=	value to which the bit should be set
whichone	=	specifies which bit should be set 0 corresponds to the lowest bit
Return	=	the changed value

util.ClearBit

Declaration:

static int ClearBit(int value, int whichone);

Deletes a bit in an integer.

Value	=	value in which the bit should be deleted
whichone	=	specifies which bit should be deleted 0 corresponds to the lowest bit
Return	=	the changed value

3.6 Java programming examples

Some brief example programs follow. The programs are available as source code and in already compiled form in the “Examples” directory.

3.6.1 AnalogExample.java

```
/** Reads the analog value every 2 seconds and travels to a
 *  position calculated from it
 *
 * */

import nanotec.io;
import nanotec.drive;
import nanotec.util;

class AnalogExample {

    /** Reads the analog value and calculates
     *  a target position from it
     *
     * */
    static int CalculateTargetPos( ){
        int pos = io.GetAnalogInput( 1 );

        pos = (pos * 2) + 1000;

        return pos;
    }

    public static void main() {

        //Configure motor
        drive.SetTargetPos(0);
        drive.SetMaxSpeed(2000);

        //Main loop
        while(true){

            io.SetLED(1);
            util.Sleep(100);

            io.SetLED(0);
            util.Sleep(1800);

            drive.StopDrive( );
            drive.SetTargetPos(CalculateTargetPos () );
            drive.StartDrive( );
        }
    }
}
```

3.6.2 DigitalExample.java

```
/** If input 1 is active, the LED is switched on
 * */

import nanotec.io;
import nanotec.util;


        class DigitalExample {

            public static void main() {

                util.Sleep(200);

                //Main loop
                while(true){

                    if( io.GetDigitalInput( ) == 65 ){
                        io.SetLED(1);
                    } else {
                        io.SetLED(0);
                    }

                }
            }
        }
```

3.6.3 TimerExample.java

```
/** Example for a timer realized with GetMillis()
 *
 *  The program lets the red LED flash
 * */

import nanotec.io;
import nanotec.util;


        class TimerExample {

            public static void main() {


                //Main loop
                while(true){
                    io.SetLED(1);
                    util.Sleep(200);

                    io.SetLED(0);
                    util.Sleep(1800);
                }
            }

        }
```

3.6.4 ConfigDriveExample.java

```
/** Configures the motor for absolute positioning
 *  and travels back and forth between 2 positions
 *          with different speeds
 * */

import nanotec.drive;
import nanotec.util;

        class ConfigDriveExample {

            public static void main() {

                //Configure motor
                drive.SetDriveMode(1);              //Positioning mode
                drive.SetMode(2);                       //Absolute positioning
                drive.SetMinSpeed(100);
                drive.SetAccelaration(2000);    //Ramp
                drive.SetCurrent(10);               //Current
                drive.SetCurrentReduction(1);  //Current for reduction

                //Main loop
                while(true){

                    drive.SetMaxSpeed(1000);        //Speed
                    drive.SetTargetPos(1000);       //Target

                    drive.StartDrive( );
                    util.Sleep(4000);           //Wait 4 seconds

                    drive.SetMaxSpeed(2000);        //Speed
                    drive.SetTargetPos(10);      //Target
                    drive.StartDrive( );
                    util.Sleep(2000);           //Wait 2 seconds


                }
            }
        }
```

3.6.5 DigitalOutput.java

```
/**Sets the outputs and sends the current status
 * via the serial interface
 *
 * */

import nanotec.io;
import nanotec.comm;
import nanotec.util;

class DigitalOutput {

    public static void main() {

        while( true ){
            io.SetDigitalOutput(0);
            comm.SendInt( GetDigitalOutput( ) );
            util.Sleep(1000);

            io.SetDigitalOutput(1);
            comm.SendInt( GetDigitalOutput( ) );
            util.Sleep(1000);

            io.SetDigitalOutput(2);
            comm.SendInt( GetDigitalOutput( ) );
            util.sleep(1000);
        }
    }
}
```

3.7 Manual translation and transfer of a program without NanoJEasy

3.7.1 Necessary tools

Introduction

Alternatively to the translation and transfer of programs from the programming environment, programs can also be translated and transferred manually.

Java SE

Java SE is the standard Java implementation from Sun. Two different versions are offered by Sun: The JRE (Java Runtime Environment) which can be used to execute a finished Java program and the JDK (Java Development Kit) that is required for the development of Java programs.

Both can be downloaded free of charge from Sun (java.sun.com). The JDK, which also contains the JRE, is necessary for developing a program for the controller. The current version is “JDK 6 Update 14”.

ejvm_linker

The ejvm_linker is a command line program which converts Java.class files in such a way that they can be processed by the controller.

It is not essential to install the program. It is helpful, however, if you enter it in the PATH variable. This means it is not necessary to enter the complete path when starting the program.

Proceed as follows for entering the program in the PATH variable:

Step	Implementation
1	Under Start -> Settings -> System driver -> System, select the “Advanced” tab.
2	Click on the <Environment variables> button.
3	Mark the variable in the “System variables” window.
3	Click on <Edit> under the “System variables” window.
4	Enter the installation path of the program under “Value of the variables”.
5	Click on <OK>.

PD4 Utility

The PD4 utility (Version 1.2 or higher required) is used for transferring firmware or program files to a controller. The program does not have to be installed, it is sufficient to execute the pd4_util.exe.

ejvm_emulator

The ejvm_emulator is used for the function test of the program on the PC. The emulator can simulate problems such as a stack overflow on the VM.

3.7.2 Translating the program

The program must be translated with the normal Java SE compiler:

`javac.exe Myprogram.java`

The result is a .class file which contains the finished program in binary form:

`Myprogram.class`

"*Myprogram*" is the placeholder for the name of your program.

3.7.3 Linking and converting a program

Overview

Before the program can be transferred to the controller, it must be linked and converted. This is carried out with the aid of the ejvm_linker.exe. Some checks are also carried out during the conversion, especially of the program size.

Starting ejvm_linker.exe without debug function

Enter:

`ejvm_linker.exe Myprogram.class Myprogram.prg`

"*Myprogram*" is the placeholder for the name of your program.

Starting ejvm_linker.exe with debug function

The program can be converted with the '-debug' switch for debug purposes. The created program then contains other additional debug information and the linker outputs detailed information. This makes the created program larger, however.

Enter:

`ejvm_linker.exe -debug Myprogram.class Myprogram.prg`

"*Myprogram*" is the placeholder for the name of your program.

Result

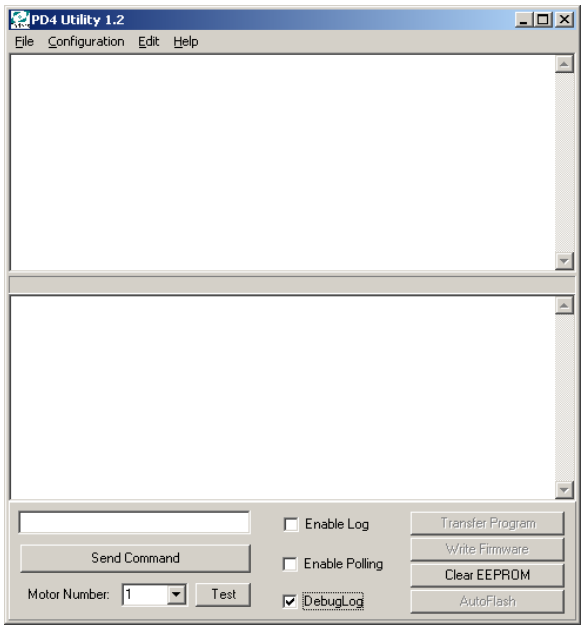
The result of the linking and conversion is a .prg file which can be loaded into the controller:

`Myprogram.prg`

3.7.4 Transferring the program to the controller

PD4 utility dialog window

The transfer to the controller is carried out with the PD4 utility:



Procedure

Proceed as follows for entering the program in the PATH variable:

Step	Implementation
1	Open the “Configuration” menu item and enter the correct COM port and a baud rate of 115,200.
2	Check that the number that appears in the “Motor Number” input field agrees with the position of the hex switch of the controller (for more details, see the manual of the controller).
3	Open the File -> Open menu item and select the .prg file of your program. The upper text field is filled out by the PD4 utility.
3	To transfer the program to the controller, click on the <Transfer Program> button.

3.7.5 Executing the program

PD4 utility

Serial commands can also be transferred to the controller with the PD4 utility. To do this, enter the desired command in the text field with the <Send Command> button.

The commands listed in the following sections are available:

(JI ... Verifying loaded Java program

This command loads the current program from the EEProm and initializes the VM. This initialization is also carried out automatically when switching on the controller and when transferring the program with the PD4 utility.

The ID of the “ECAFFE01” program is received as the response to the command. See also Section *2.8.4 Verifying loaded Java* program.

(JA ... Starting a loaded Java program

This command starts the program. (JA+ is received as the response if the program was started successfully or (JA- if the program could not be started (no valid or no program at all installed on the controller). See also Section *2.8.2 Starting a loaded* Java program.

(JS ... Stopping the running Java program

This command stops the program.

(JS+ is received as the response if the program was stopped successfully or (JS- if the program was already ended. See also Section *2.8.3 Stopping the running* Java program.

(JB ... Automatically starting the Java program when switching on the controller

This command can be used to determine whether the program is started automatically when the controller is switched on:

- (JB=1 the program is started automatically.
- (JB=0 the program is not started automatically.

See also Section *2.8.5 Automatically starting the Java program when switching* on the controller.

(JE ... Reading out error of the Java program

This command reads out the last error:

- ERROR_NOT_NATIVE 1
- ERROR_FUNCTION_PARAMETER_TYPE 2
- ERROR_FUNCTION_NOT_FOUND 3
- ERROR_NOT_LONG 4
- ERROR_UNKNOWN_OPCODE 5
- ERROR_TOO_MANY_PARAMS 6
- ERROR_NO_MAIN_METHOD 7
- ERROR_CP_OUT_OF_RANGE 8
- ERROR_LOCAL_VAR_OUT_OF_RANGE 9
- ERROR_NOT_AN_VAR_IDX A
- ERROR_VAR_IS_NO_INT B
- ERROR_STACK_OVERFLOW C
- ERROR_STACK_UNDERFLOW D
- ERROR_HEAP_OVERFLOW E
- ERROR_HEAP_UNDERFLOW F
- ERROR_FRAME_OVERLOW 10
- ERROR_UNKNOWN_DATATYPE 11
- ERROR_LOCAL_VAR_OVERFLOW 12

See also Section *2.8.6 Reading out error of* the Java program and *3.8 Possible Java* error messages.

(JW ... Reading out warning

This command reads out the last warning:

WARNING_FUNCTION_NOT_SUPPORTED 1

To display the outputs of the program, the checkmark must be set against “Debug Log” (see “DigitalOutput.java” program example). See also Section *2.8.7 Reading out the warning* of the Java program.

3.8 Possible Java error messages

Meaning of the error messages

The error messages read out with the “(JE” command have the following meaning:

Index	Error message	Meaning
1	ERROR_NOT_NATIVE	This command is not supported by the controller.
2	ERROR_FUNCTION_PARAMETER_TYPE	The transfer parameter of a function has the wrong type (e.g. “float” instead of “int”).
3	ERROR_FUNCTION_NOT_FOUND	An unknown function has been called up. Check that all files have been included. See also Section *3.4.3 Integrated commands* (Include Manager).
4	ERROR_NOT_LONG	An incorrect data type is being used (should be “long”).
5	ERROR_UNKNOWN_OPCODE	A Java function that is not supported is being called up (e.g. “new”).
6	ERROR_TOO_MANY_PARAMS	The number of parameters in the call-up of a function is not correct.
7	ERROR_NO_MAIN_METHOD	The “public static void main()” function is missing.
8	ERROR_CP_OUT_OF_RANGE	Memory error: Check that all files have been included. See also Section *3.4.3 Integrated commands* (Include Manager).
9	ERROR_LOCAL_VAR_OUT_OF_RANGE	Memory error: Check that all files have been included. See also Section *3.4.3 Integrated commands* (Include Manager).
A	ERROR_NOT_AN_VAR_IDX	Memory error: Check that all files have been included. See also Section *3.4.3 Integrated commands* (Include Manager).

Index	Error message	Meaning
B	ERROR_VAR_IS_NO_INT	An incorrect data type is being used (should be “int”).
C	ERROR_STACK_OVERFLOW	Stack overflow: Too many function calls have been nested within one another (possibly recursion too deep).
D	ERROR_STACK_UNDERFLOW	Stack underflow: Check that all files are included. See also Section *3.4.3 Integrated commands* (Include Manager).
E	ERROR_HEAP_OVERFLOW	Heap overflow: Too many function calls have been nested within one another (possibly recursion too deep).
F	ERROR_HEAP_UNDERFLOW	Heap underflow: Check that all files have been included. See also Section *3.4.3 Integrated commands* (Include Manager).
10	ERROR_FRAME_OVERLOW	Frame overflow: Too many class call-ups have been used.
11	ERROR_UNKNOWN_DATATYPE	An unknown data type is used.
12	ERROR_LOCAL_VAR_OVERFLOW	Memory error: Check that all files have been included. See also Section *3.4.3 Integrated commands* (Include Manager).

See also Section *2.8.6 Reading out error of* the Java program and Section *3.7.5 Executing the* program.

4 Programming via the COM interface

4.1 Overview

About this chapter

This chapter contains an overview of the COM interface for programming the Nanotec stepper motor positioning controls.

Operating systems and NanoPro versions

The functions required for serial communication with the stepper motor positioning controls are currently only written for the Windows operating system and its derivates (x64).

This documentation is valid from NanoPro version 0.51.0.41 and SDK version 0.51.0.41.

Preconditions

To develop a program for controlling the stepper motor positioning controls, the following preconditions must be fulfilled:

- Programming knowledge is required.
- The SDK (Software Development Kit) for “NanoPro” should be installed. The PD4I.dll command is registered on its installation.
- The .net framework 2.0 must be installed.

In order to try out “Office examples” with Excel, the VBA add-on for Excel must be installed. For the smooth interaction of the individual components, MS-Office 2003 and higher should be used.

Programming environments

VBA (Visual Basic) or any other high language IDE such as Visual Studio, for example, can be used as the programming environment. A Visual Studio project file is included with the examples.

4.2 Command overview

A list of the commands for programming via the COM interface can be found below:

4.3 Description of the functions

Methods

There are two categories of methods:

- One is the so-called 'Set' method which hands over information to the controller.
- The other is the 'Get' method that fetches the information from the controller. The value returned in the 'Set' method can be used to check that the information has also been sent to the controller.

Calling up the status of the objects

Information on the status of the object can be called up explicitly after every call-up of the method with the following functions:

- Errorflag this function returns the error status
- ErrorNumber this function returns the error number
- ErrorMessageString this function returns a description of the error

4.3.1 General functions

Errorflag

Definition:

bool **Errorflag**

This function can be used to query whether an error has occurred.

ErrorNumber

Definition:

int **ErrorNumber**

This function can be used to query the error number.

ErrorMessageString

Definition:

string **ErrorMessageString**

This function can be used to query the description of the error.

SerialPorts

Definition:

string[] **SerialPorts**

This function can be used to query a list of available serial interfaces of the computer system.

Use:

string[ ] ports = SelectedPort

SelectedPort

Definition:

string **SelectedPort**

This function can be used to set or query the serial interface to be used.

Use:

string port = SelectedPort

port = “COM40”

Selectedport = port

Baud rate

Definition:

int **Baudrate**

This function can be used to set or query the transfer rate.

GetAvailableMotorAddresses

Definition:

IList<int> **GetAvailableMotorAddresses**

This function can be used to query the list of available motor addresses.

MotorAddresse

Definition:

int **MotorAddresse**

This function is used to set or query the motor address of the COM object previously created.

4.3.2 Status functions for older motors

GetStatusByte

Definition:

byte **GetStatusByte()**

This function can be used to query the status byte of the controller.

The function corresponds to the serial command '$', see command *2.5.21 Reading out* the status.

IsMotorReady()

Definition:

bool **IsMotorReady()**

This function returns true if the motor is ready.

IsAtReferencePosition()

Definition:

bool **IsAtReferencePosition()**

This function returns true if the motor is at the reference position.

HasPositionError

Definition:

bool **HasPositionError()**

This function returns true if the motor has a position error.

HasEndedTravelProfileAndStartInputStillActive

Definition:

bool **HasEndedTravelProfileAndStartInputStillActive()**

This function returns true if the travel profile has ended and the start input signal is still active.

IsPositionModeActive

Definition:

bool **IsPositionModeActive()**

This function returns true if the positioning mode in the controller is active.

IsSpeedModeActive

Definition:

bool **IsSpeedModeActive()**

This function returns true if the speed mode in the controller is active.

IsFlagPositionModeActive

Definition:

bool **IsFlagPositionModeActive()**

This function returns true if the flag positioning mode in the controller is active.

IsClockDirectionModeActive

Definition:

bool **IsFlagPositionModeActive()**

This function returns true if the clock direction mode in the controller is active.

IsJoyStickModeActive

Definition:

bool **IsJoyStickModeActive()**

This function returns true if the joystick mode in the controller is active.

IsAnalogModeActive

Definition:

bool **IsAnalogModeActive()**

This function returns true if the analog mode in the controller is active.

IsTorqueModeActive

Definition:

bool **IsTorqueModeActive()**

This function returns true if the torque mode in the controller is active.

4.3.3 Motor control functions for older motors

ResetPositionError

Definition:

bool **ResetPositionError()**

This function can be used to reset the position error.

The value returned by the function can be used to check that the command was correctly recognized by the controller.

The function corresponds to the serial command 0x44.

StartTravelProfile

Definition:

bool **StartTravelProfile()**

This function can be used to start the travel profile.

The value returned by the function can be used to check that the command was correctly recognized by the controller.

The function corresponds to the serial command 0x41.

StopTravelProfile

Definition:

bool **StopTravelProfile()**

This function can be used to stop the travel profile.

The value returned by the function can be used to check that the command was correctly recognized by the controller.

The function corresponds to the serial command 0x53.

IncreaseFrequency

Definition:

bool **IncreaseFrequency()**

This function increases the frequency of the motor.

The value returned by the function can be used to check that the command was correctly recognized by the controller.

The function corresponds to the serial command 0x2b.

DecreaseFrequency

Definition:

bool **DecreaseFrequency()**

This function decreases the frequency of the motor.

The value returned by the function can be used to check that the command was correctly recognized by the controller.

The function corresponds to the serial command 0x2d.

TriggerOn

Definition:

bool **TriggerOn()**

This function switches the trigger of the motor on.

The value returned by the function can be used to check that the command was correctly recognized by the controller.

The function corresponds to the serial command 0x54.

StoreRecord

Definition:

bool **StoreRecord(***int recordNumber***)**

This function saves the parameters previously set in the record with the number of the value handed over.

The value returned by the function can be used to check that the command was correctly recognized by the controller.

The function corresponds to the serial command 0x3e.

WriteCurrentReduction

Definition:

bool **WriteCurrentReduction(***int currentReduction***)**

This function sets the current of the current reduction at standstill in percent. Like the phase current, this current is relative to the end value. Values above 100 should be avoided.

The value returned by the function can be used to check that the command was correctly recognized by the controller.

The function corresponds to the serial command 0x72.

WritePhaseCurrent

Definition:

bool **WritePhaseCurrent(***int phaseCurrent***)**

This function sets the phase current in percent. Values above 100 should be avoided.

The value returned by the function can be used to check that the command was correctly recognized by the controller.

The function corresponds to the serial command 0x69.

WriteFinalySendStatus

Definition:

bool **WriteFinalySendStatus(***bool value***)**

This function switches the independent sending of a status at the end of a travel:

- value = 0, sending switched off
- value = 1, sending switched on

The value returned by the function can be used to check that the command was correctly recognized by the controller.

The function corresponds to the serial command 0x4a.

WriteAnalogueMin

Definition:

bool **WriteAnalogueMin(***double min***)**

This function sets the lower voltage level of the motor.

The value returned by the function can be used to check that the command was correctly recognized by the controller.

The function corresponds to the serial command 0x51.

WriteAnalogueMax

Definition:

bool **WriteAnalogueMax(***double max***)**

This function sets the upper voltage level of the motor.

The value returned by the function can be used to check that the command was correctly recognized by the controller.

The function corresponds to the serial command 0x52.

WriteErrorCorrectionRecord

Definition:

bool **WriteErrorCorrectionRecord(***string value, bool enabled***)**

This function switches the automatic error correction of the controller on:

- Value = the record which is carried out in the event of error
- enabled = 0, error correction switched off
- enabled = 1, error correction switched on

The value returned by the function can be used to check that the command was correctly recognized by the controller.

The function corresponds to the serial command 0x46.

WriteToleranceWidth

Definition:

bool **WriteToleranceWidth(***int value***)**

This function sets the tolerance width of the controller.

The value returned by the function can be used to check that the command was correctly recognized by the controller.

The function corresponds to the serial command 0x58.

WriteSwingOutTime

Definition:

bool **WriteSwingOutTime(***int value, bool enabled***)**

This function sets the swing out time of the controller:

- Value = values of the swing out time
- enabled = 0, swing out time switched off
- enabled = 1, swing out time switched on

The value returned by the function can be used to check that the command was correctly recognized by the controller.

The function corresponds to the serial command 0x4F.

WriteReverseEncoderRotatingDirection

Definition:

bool **WriteReverseEncoderRotatingDirection(***bool value***)**

This function sets the reversal of the encoder direction.

The value returned by the function can be used to check that the command was correctly recognized by the controller.

The function corresponds to the serial command 0x71.

SetAddress

Definition:

bool **SetAddress(***int newMotoraddress***)**

This function sets a new motor address.

The value returned by the function can be used to check that the command was correctly recognized by the controller.

The function corresponds to the serial command 0x6d.

SetReverseClearance

Definition:

bool **SetReverseClearance(***int reverseClearence***)**

This function sets the reverse clearance in steps.

The value returned by the function can be used to check that the command was correctly recognized by the controller.

The function corresponds to the serial command 0x7a.

ResetCounter

Definition:

bool **ResetCounter()**

This function sets the position counter to the value 0.

The value returned by the function can be used to check that the command was correctly recognized by the controller.

The function corresponds to the serial command 0x63.

WriteStartFrequency

Definition:

bool **WriteStartFrequency(***double frequency***)**

This function sets the start frequency of the motor.

The value returned by the function can be used to check that the command was correctly recognized by the controller.

The function corresponds to the serial command 0x75.

WriteRamp

Definition:

bool **WriteRamp(***int ramp***)**

This function sets the ramp of the motor.

The value returned by the function can be used to check that the command was correctly recognized by the controller.

The function corresponds to the serial command 0x62.

WriteNumberOfPasses

Definition:

bool **WriteNumberOfPasses(***int numberOfPasses***)**

This function sets the number of repetitions for the motor.

The value returned by the function can be used to check that the command was correctly recognized by the controller.

The function corresponds to the serial command 0x57.

WriteNormalFrequency

Definition:

bool **WriteNormalFrequency(***double frequency***)**

This function sets the target frequency of the motor.

The value returned by the function can be used to check that the command was correctly recognized by the controller.

The function corresponds to the serial command 0x6f.

WriteNextOperation

Definition:

bool **WriteNextOperation(***int operationNumber***)**

This function sets the next record to be carried out.

The value returned by the function can be used to check that the command was correctly recognized by the controller.

The function corresponds to the serial command 0x4e.

WriteSteps

Definition:

bool **WriteSteps(***int steps***)**

This function sets the number of steps of the motor to be carried out.

The value returned by the function can be used to check that the command was correctly recognized by the controller.

The function corresponds to the serial command 0x73.

WriteMaximumFrequency

Definition:

bool **WriteMaximumFrequency(***double frequency***)**

This function sets the maximum frequency of the motor.

The value returned by the function can be used to check that the command was correctly recognized by the controller.

The function corresponds to the serial command 0x6e.

ReadChangeDirection

Definition:

bool ***ReadChangeDirection(****int operationNumber***)**

This function reads the change of direction of the motor:

- *changeDirection* = 0, change of direction switched off
- *changeDirection* = 1, change of direction switched on

Here the *operationNumber* parameter is the record number (travel profile) that should be read from.

The function uses the serial command 0x5a.

WriteChangeDirection

Definition:

bool **WriteChangeDirection(***bool changeDirection***)**

This function switches on the change of direction of the motor:

- *changeDirection* = 0, change of direction switched off
- *changeDirection* = 1, change of direction switched on

The value returned by the function can be used to check that the command was correctly recognized by the controller.

The function corresponds to the serial command 0x74.

WriteBreak

Definition:

bool **WriteBreak(***double brakTime***)**

This function sets the pause time of the motor in milliseconds.

The value returned by the function can be used to check that the command was correctly recognized by the controller.

The function corresponds to the serial command 0x50.

ReadRecord

Definition:

int **ReadRecord(***int operationNumber***)**

This function reads a record (data record) of the motor.

Here the *operationNumber* parameter is the record number (travel profile) that should be read from.

The function uses the serial command 0x5a.

ReadAddress

Definition:

bool **ReadAddress(***int operationNumber***)**

This function reads the motor address of the connected motor.

Here the *operationNumber* parameter is the record number (travel profile) that should be read from.

The function uses the serial command 0x4d.

Attention:
When using this command, only one motor should be connected.

ReadCounter

Definition:

bool **ReadCounter(***double operationNumber***)**

This function reads the current position of the motor.

Here the *operationNumber* parameter is the record number (travel profile) that should be read from.

The function uses the serial command 0x43.

ReadStartFrequency

Definition:

double **ReadStartFrequency(***int operationNumber***)**

This function reads the start frequency of the motor.

Here the *operationNumber* parameter is the record number (travel profile) that should be read from.

The function uses the serial command 0x5a.

ReadRamp

Definition:

int **ReadRamp(***int operationNumber***)**

This function reads the ramp of the motor.

Here the *operationNumber* parameter is the record number (travel profile) that should be read from.

The function uses the serial command 0x5a.

ReadNumberOfPasses

Definition:

int **ReadNumberOfPasses(***int operationNumber***)**

This function reads the number of runs of the motor.

Here the *operationNumber* parameter is the record number (travel profile) that should be read from.

The function uses the serial command 0x5a.

ReadNormalFrequency

Definition:

double **ReadNormalFrequency(***int operationNumber***)**

This function reads the target frequency of the motor.

Here the *operationNumber* parameter is the record number (travel profile) that should be read from.

The function uses the serial command 0x5a.

ReadNextOperation

Definition:

int **ReadNextOperation(***int operationNumber***)**

This function reads the number of the next record (travel profile) to be carried out.

Here the *operationNumber* parameter is the record number (travel profile) that should be read from.

The function uses the serial command 0x5a.

ReadSteps

Definition:

int **ReadSteps(***int operationNumber***)**

This function reads the number of the steps set in the controller of the motor.

Here the *operationNumber* parameter is the record number (travel profile) that should be read from.

The function uses the serial command 0x5a.

ReadMemory

Definition:

int **ReadMemory(***int speicherAddress***)**

This function reads a value from a specific memory address. The memory address is handed over as a parameter.

The function uses the serial command 0x5a.

ReadMaximumFrequency

Definition:

int **ReadMaximumFrequency(***int operationNumber***)**

This function reads the maximum frequency of the motor.

Here the *operationNumber* parameter is the record number (travel profile) that should be read from.

The function uses the serial command 0x5a.

ReadBreak

Definition:

int **ReadBreak(***int operationNumber***)**

This function reads the pause time of the motor in milliseconds.

Here the *operationNumber* parameter is the record number (travel profile) that should be read from.

The function uses the serial command 0x5a.

ReadCurrentReduction

Definition:

int **ReadCurrentReduction(***int operationNumber***)**

This function reads the current reduction at standstill in percent.

Here the *operationNumber* parameter is the record number (travel profile) that should be read from.

The function uses the serial command 0x5a.

ReadPhaseCurrent

Definition:

int **ReadPhaseCurrent(***int operationNumber***)**

This function reads the phase current of the motor in percent.

Here the *operationNumber* parameter is the record number (travel profile) that should be read from.

The function uses the serial command 0x5a.

GetOperationMode

Definition:

int **GetOperationMode()**

This function reads the operation mode of the motor that is currently active:

- Return 1 corresponds to positioning mode
- Return = 2 corresponds to speed mode
- Return = 3 corresponds to flag positioning mode
- Return = 4 corresponds to clock direction mode

The function uses the serial command 0x21.

WriteStepMode

Definition:

bool **WriteStepMode(***int stepMode***)**

This function sets the step mode of the motor:

- stepMode = 0 corresponds to a full step
- stepMode = 1 corresponds to half of a step
- stepMode = 2 corresponds to a quarter of a step
- stepMode = 3 corresponds to a fifth of a step
- stepMode = 4 corresponds to an eighth of a step
- stepMode = 5 corresponds to a tenth of a step
- stepMode = 6 corresponds to a sixteenth of a step
- stepMode = 7 corresponds to a thirty-secondth of a step
- stepMode = 8 corresponds to a sixty-fourth of a step
- stepMode = 9 corresponds to adaptive microstep

The value returned by the function can be used to check that the command was correctly recognized by the controller.

The function uses the serial command 0x67.

WriteInternalNormalRunBehavior

Definition:

bool **WriteInternalNormalRunBehavior(***int normalRunBehavior***)**

This function sets the limit switch behavior during the normal internal run of the motor:

- *normalRunBehavior* = 0 corresponds to disable
- *normalRunBehavior* = 1 corresponds to free backwards
- *normalRunBehavior* = 2 corresponds to free forwards
- *normalRunBehavior* = 3 corresponds to stop

The value returned by the function can be used to check that the command was correctly recognized by the controller.

The function uses the serial command 0x65.

WriteInternalReferenceRunBehavior

Definition:

bool **WriteInternalReferenceRunBehavior(***int refrenceBehavior***)**

This function sets the limit switch behavior during the internal reference run of the motor:

- *refrenceBehavior* = 0 corresponds to free backwards
- *refrenceBehavior* = 1 corresponds to free forwards

The value returned by the function can be used to check that the command was correctly recognized by the controller.

The function uses the serial command 0x65.

WriteExternalSwitchType

Definition:

int **WriteExternalSwitchType (***int switchType***)**

This function sets the external limit switch type of the motor:

- *switchType* = 0 corresponds to opener
- *switchType* = 1 corresponds to closer

The value returned by the function can be used to check that the command was correctly recognized by the controller.

The function uses the serial command 0x6c.

WriteExternalNormalRunBehavior

Definition:

bool **WriteExternalNormalRunBehavior(***int normalRunBehavior***)**

This function sets the limit switch behavior during an external normal run of the motor:

- *normalRunBehavior* = 0 corresponds to disable
- *normalRunBehavior* = 1 corresponds to free backwards
- *normalRunBehavior* = 2 corresponds to free forwards
- *normalRunBehavior* = 3 corresponds to stop

The value returned by the function can be used to check that the command was correctly recognized by the controller.

The function uses the serial command 0x65.

WriteExternalReferenceRunBehavior

Definition:

bool **WriteExternalReferenceRunBehavior(***int refrenceBehavior***)**

This function sets the limit switch behavior during an external reference run of the motor:

- *refrenceBehavior* = 0 corresponds to free backwards
- *refrenceBehavior* = 1 corresponds to free forwards

The value returned by the function can be used to check that the command was correctly recognized by the controller.

The function uses the serial command 0x65.

WriteOperationType

Definition:

bool **WriteOperationType(***int operationType***)**

This function sets the operation type of the motor:

- *operationType* = 0 corresponds to relative; depends on the operation mode
- *operationType* = 1 corresponds to absolute; depends on the operation mode
- *operationType* = 3 corresponds to an internal reference run
- *operationType* = 4 corresponds to an external reference run

The value returned by the function can be used to check that the command was correctly recognized by the controller.

The function uses the serial command 0x70.

ReadOperationType

Definition:

int **ReadOperationType(***int operationNumber***)**

This function reads the selected operation type of the motor.

- *Return* = 0 corresponds to relative; depends on the operation mode
- *Return* = 1 corresponds to absolute; depends on the operation mode
- *Return* = 3 corresponds to an internal reference run
- *Return* = 4 corresponds to an external reference run

Here the *operationNumber* parameter is the record number (travel profile) that should be read from.

The function uses the serial command 0x5a.

WriteDirection

Definition:

bool **WriteDirection(***int direction***)**

This function sets the direction of rotation of the motor:

- direction = 0 corresponds to left
- direction = 1 corresponds to right

The value returned by the function can be used to check that the command was correctly recognized by the controller.

The function uses the serial command 0x64.

ReadDirection

Definition:

int **ReadDirection(***int operationNumber***)**

This function reads the direction of rotation of the motor:

- *Return* = 0 corresponds to left
- *Return* = 1 corresponds to right

Here the *operationNumber* parameter is the record number (travel profile) that should be read from.

The function uses the serial command 0x5a.

ReadReverseClearance

Definition:

int **ReadReverseClearance()**

This function reads the reverse clearance of the motor.

The function uses the serial command 0x5a.

4.3.4 Status functions for newer motors

GetNewStatus

Definition:

byte **GetNewStatus()**

This function can be used to query the status of the controller.

IsNewMotorReady

Definition:

bool **IsNewMotorReady()**

This function returns true if the motor is ready.

IsAtNewReferencePosition

Definition:

bool **IsAtNewReferencePosition()**

This function returns true if the motor is at the reference position.

HasNewPositionError

Definition:

bool **HasNewPositionError()**

This function returns true if the motor has a position error.

HasNewEndedTravelProfileAndStartInputStillActive

Definition:

bool **HasNewEndedTravelProfileAndStartInputStillActive()**

This function returns true if the travel profile has ended and the start input signal is still active.

IsNewPositionModeActive

Definition:

bool **IsNewPositionModeActive()**

This function returns true if the positioning mode in the controller is active.

IsNewSpeedModeActive

Definition:

bool **IsNewSpeedModeActive()**

This function returns true if the speed mode in the controller is active.

IsNewFlagPositionModeActive

Definition:

bool **IsNewFlagPositionModeActive()**

This function returns true if the flag positioning mode in the controller is active.

IsNewClockDirectionModeActive

Definition:

bool **IsNewFlagPositionModeActive()**

This function returns true if the clock direction mode in the controller is active.

IsNewNewJoyStickModeActive

Definition:

bool **IsNewJoyStickModeActive()**

This function returns true if the joystick mode in the controller is active.

IsNewAnalogModeActive

Definition:

bool **IsNewAnalogModeActive()**

This function returns true if the analog mode in the controller is active.

IsNewTorqueModeActive

Definition:

bool **IsNewTorqueModeActive()**

This function returns true if the torque mode in the controller is active.

4.3.5 Motor control functions for newer motors

SetNewSoftware

Definition:

bool **SetNewSoftware(***bool value***)**

This function sets the information that a corresponding (newer) command set for the motor should be selected for the COM object.

0 corresponds to yes

1 corresponds to no

ResetAllSettings

Definition:

bool **ResetAllSettings()**

This function sets the settings of the controller back to default values (factory default settings).

The function corresponds to the serial command '~', see command *2.5.28 Carrying out* an EEPROM reset.

GetNewVersion

Definition:

string **GetNewVersion()**

This function reads the version text from the controller.

The function corresponds to the serial command 'v', see command *2.5.22 Reading out* the firmware version.

ResetNewPositionError

Definition:

bool **ResetNewPositionError()**

This function can be used to reset the position error.

The value returned by the function can be used to check that the command was correctly recognized by the controller.

The function corresponds to the serial command 'D', see command *2.5.14 Resetting the* position error.

SetNewSendStatusWhenCompleted

Definition:

bool **SetNewSendStatusWhenCompleted(***bool sendStatus***)**

This function switches the independent sending of a status at the end of a travel:

- *sendStatus* = 0, sending switched off
- *sendStatus* = 1, sending switched on

The value returned by the function can be used to check that the command was correctly recognized by the controller.

The function corresponds to the serial command 'J', see command *2.5.29 Setting automatic sending of the* status.

GetNewSendStatusWhenCompleted

Definition:

bool **GetNewSendStatusWhenCompleted()**

This function reads whether the independent sending of a status at the end of a travel is switched on:

- *Return* = 0, sending switched off
- *Return* = 1, sending switched on

The function corresponds to the serial command 'J', see command *2.5.29 Setting automatic sending of the* status.

GetNewPosition

Definition:

int **GetNewPosition()**

This function reads the current position of the motor.

The function corresponds to the serial command 'C', see command *2.5.17 Reading out* the position.

ResetNewPosition

Definition:

bool ResetNewPosition**()**

This function sets the position counter to the value 0.

The value returned by the function can be used to check that the command was correctly recognized by the controller.

The function corresponds to the serial command 'c', see command *2.5.18 Resetting the* position.

StartNewTravelProfile

Definition:

bool **StartNewTravelProfile()**

This function can be used to start the travel profile.

The value returned by the function can be used to check that the command was correctly recognized by the controller.

The function corresponds to the serial command 'A', see command *2.6.1 Starting the* motor.

StopNewTravelProfile

Definition:

bool **StopNewTravelProfile()**

This function can be used to stop the travel profile.

The value returned by the function can be used to check that the command was correctly recognized by the controller.

The function corresponds to the serial command 'S', see command *2.6.2 Stopping a* motor.

IncreaseNewFrequency

Definition:

bool **IncreaseNewFrequency()**

This function increases the frequency of the motor.

The value returned by the function can be used to check that the command was correctly recognized by the controller.

The function corresponds to the serial command '+', see command *2.7.7 Increasing the* speed.

DecreaseNewFrequency

Definition:

bool **DecreaseNewFrequency()**

This function decreases the frequency of the motor.

The value returned by the function can be used to check that the command was correctly recognized by the controller.

The function corresponds to the serial command '-', see command *2.7.8 Reducing the* speed.

NewTriggerOn

Definition:

bool **NewTriggerOn()**

This function switches the trigger of the motor on.

The value returned by the function can be used to check that the command was correctly recognized by the controller.

The function corresponds to the serial command 'T', see command *2.7.9 Actuating the* trigger.

ChooseNewRecord

Definition:

bool **ChooseNewRecord(***int recordNumber***)**

This function reads a specific record (travel profile) of the motor.

The *recordNumber* parameter is the record number (travel profile) that should be read.

The value returned by the function can be used to check that the command was correctly recognized by the controller.

The function corresponds to the serial command 'y', see command *2.6.3 Loading a record from* the EEPROM.

SetNewRecord

Definition:

bool **SetNewRecord(***int recordNumber***)**

This function writes a new specific record (travel profile) of the motor.

The *recordNumber* parameter is the record number (travel profile) that should be written.

The value returned by the function can be used to check that the command was correctly recognized by the controller.

The function corresponds to the serial command '>', see command *2.6.5 Saving a* record.

SetNewPlay

Definition:

bool **SetNewPlay(***int play***)**

This function sets the dead range of the motor.

The value returned by the function can be used to check that the command was correctly recognized by the controller.

The function corresponds to the serial command '=', see command *2.7.1 Setting the dead* range for the joystick mode.

GetNewPlay

Definition:

int **GetNewPlay()**

This function reads the dead range of the motor.

The function corresponds to the serial command '=', see command *2.7.1 Setting the dead* range for the joystick mode.

SetNewSoftwareFilter

Definition:

bool **SetNewSoftwareFilter(***int softwareFilter***)**

This function sets the filter of the motor.

The value returned by the function can be used to check that the command was correctly recognized by the controller.

The function corresponds to the serial command 'f', see command *2.7.2 Setting the filter for the analog* and joystick modes.

GetNewSoftwareFilter

Definition:

int **GetNewSoftwareFilter()**

This function reads the filter of the motor.

The function corresponds to the serial command 'f', see command *2.7.2 Setting the filter for the analog* and joystick modes.

SetNewStepMode

Definition:

bool **SetNewStepMode(***int stepMode***)**

This function sets the step mode of the motor:

- stepMode = 0 corresponds to a full step
- stepMode = 1 corresponds to half of a step
- stepMode = 2 corresponds to a quarter of a step
- stepMode = 3 corresponds to a fifth of a step
- stepMode = 4 corresponds to an eighth of a step
- stepMode = 5 corresponds to a tenth of a step
- stepMode = 6 corresponds to a sixteenth of a step

- stepMode = 7 corresponds to a thirty-secondth of a step
- stepMode = 8 corresponds to a sixty-fourth of a step
- stepMode = 9 corresponds to adaptive microstep

The value returned by the function can be used to check that the command was correctly recognized by the controller.

The function corresponds to the serial command 'g', see command *2.5.3 Setting the* step mode.

GetNewStepMode

Definition:

int **GetNewStepMode()**

This function reads the step mode of the motor:

- Return = 0 corresponds to full step
- Return = 1 corresponds to half of a step
- Return = 2 corresponds to a quarter of a step
- Return = 3 corresponds to a fifth of a step
- Return = 4 corresponds to an eighth of a step
- Return = 5 corresponds to a tenth of a step
- Return = 6 corresponds to a sixteenth of a step
- Return = 7 corresponds to a thirty-secondth of a step
- Return = 8 corresponds to a sixty-fourth of a step
- Return = 9 corresponds to adaptive microstep

The function corresponds to the serial command 'g', see command *2.5.3 Setting the* step mode.

SetNewMotorAddress

Definition:

bool **SetNewMotorAddress(***int motorNumber***)**

This function sets a new motor address.

The value returned by the function can be used to check that the command was correctly recognized by the controller.

The function corresponds to the serial command 'm', see command *2.5.4 Setting the* motor address.

GetNewMotorAddress

Definition:

int **GetNewMotorAddress()**

This function reads the motor address.

The function corresponds to the serial command 'M', see command *2.5.20 Reading out* the motor address.

Attention:
For using this command, only one motor must be connected.

GetNewErrorAddress

Definition:

int **GetNewErrorAddress()**

This function reads the error address at which the last error code is found from the controller.

The function corresponds to the serial command 'E', see command *2.5.15 Reading out* the error memory.

GetNewError

Definition:

int **GetNewError(***int errorAddress***)**

This function reads the error (status) to the address handed over.

The function corresponds to the serial command 'E', see command *2.5.15 Reading out* the error memory.

SetNewEnableAutoCorrect

Definition:

bool **SetNewEnableAutoCorrect(***string recordNumber, bool autocorrct***)**

This function switches on the automatic error correction of the motor.

The *recordNumber* parameter is the record number (travel profile) for which the error correction should be switched on.

The value returned by the function can be used to check that the command was correctly recognized by the controller.

The function corresponds to the serial command 'F', see command *2.5.10 Setting the record for* auto correction.

SetNewSwingOutTime

Definition:

bool **WriteSwingOutTime(***int value***)**

This function sets the swing out time of the controller.

The value returned by the function can be used to check that the command was correctly recognized by the controller.

The function corresponds to the serial command 'O', see command *2.5.12 Setting the* settling time.

GetNewSwingOutTime

Definition:

int **GetNewSwingOutTime()**

This function reads the swing out time of the controller.

The function corresponds to the serial command 'O', see command *2.5.12 Setting the* settling time.

SetNewNextRecord

Definition:

bool **SetNextRecord(***int recordNumber***)**

This function sets the next record to be carried out.

The value returned by the function can be used to check that the command was correctly recognized by the controller.

The function corresponds to the serial command 'N', see command *2.6.19 Setting the* continuation record.

GetNewNextRecord

Definition:

int **GetNextRecord(***int recordNumber***)**

This function reads the next record to be carried out.

Here the *recordNumber* parameter is the record number (travel profile) for which the next record should be read.

The function corresponds to the serial command 'N', see command *2.6.19 Setting the* continuation record.

SetNewOperationMode

Definition:

bool **SetNewOperationMode(***int operationMode***)**

This function sets the operation mode:

- *operationMode* = 1 corresponds to positioning mode
- *operationMode* = 2 corresponds to speed mode
- *operationMode* = 3 corresponds to flag positioning mode
- *operationMode* = 4 corresponds to clock direction mode
- *operationMode* = 5 corresponds to analog mode
- *operationMode* = 6 corresponds to joystick mode
- *operationMode* = 7 corresponds to analog positioning mode
- *operationMode* = *9* corresponds to torque mode

The value returned by the function can be used to check that the command was correctly recognized by the controller.

The function corresponds to the serial command '!', see command *2.5.5 Setting the* motor mode.

GetNewOperationMode

Definition:

int **GetNewOperationMode()**

This function reads the operation mode that is currently active:

- Return 1 corresponds to positioning mode
- Return = 2 corresponds to speed mode
- Return = 3 corresponds to flag positioning mode
- Return = 4 corresponds to clock direction mode
- Return = 5 corresponds to analog mode
- Return = 6 corresponds to joystick mode

- Return = 7 corresponds to analog positioning mode
- Return = 9 corresponds to torque mode

The function corresponds to the serial command '!', see command *2.5.5 Setting the* motor mode.

SetNewPhaseCurrent

Definition:

bool **SetNewDirection(***int phaseCurrent***)**

This function sets the phase current in percent. Values above 100 should be avoided.

The function corresponds to the serial command 'r', see command *2.5.2 Setting the phase current* at a standstill.

GetNewPhaseCurrent

Definition:

int **GetNewPhaseCurrent()**

This function reads the phase current in percent.

The function corresponds to the serial command 'i', see command *2.5.1 Setting the* phase current.

SetNewCurrentReduction

Definition:

bool **SetNewCurrentReduction(***int currentReduktion***)**

This function sets the current of the current reduction at standstill in percent. Like the phase current, this current is relative to the end value. Values above 100 should be avoided.

The function corresponds to the serial command 'r', see command *2.5.2 Setting the phase current* at a standstill.

GetNewCurrentReduction

Definition:

int **GetNewCurrentReductiont()**

This function sets the current of the current reduction at standstill in percent.

The function corresponds to the serial command 'r', see command *2.5.2 Setting the phase current* at a standstill.

SetNewLimitSwitchType

Definition:

int **SetNewLimitSwitchType(***int switchType***)**

This function sets the external limit switch type of the motor:

- *switchType* = 0 corresponds to opener
- *switchType* = 1 corresponds to closer

The value returned by the function can be used to check that the command was correctly recognized by the controller.

The function uses the serial command 'e', see command *2.5.7 Setting the* limit switch type.

GetNewLimitSwitchType

Definition:

int **GetNewLimitSwitchType()**

This function reads the external limit switch type of the motor:

- *switchType* = 0 corresponds to opener
- *switchType* = 1 corresponds to closer

The function uses the serial command 'e', see command *2.5.7 Setting the* limit switch type.

SetNewReverseClearance

Definition:

bool **SetNewReverseClearance(***int reverseClearence***)**

This function sets the reverse clearance in steps.

The value returned by the function can be used to check that the command was correctly recognized by the controller.

The function corresponds to the serial command 'z', see command *2.5.31 Setting the* reverse clearance.

GetNewReverseClearance

Definition:

int **GetNewReverseClearance()**

This function reads the reverse clearance in steps.

The function corresponds to the serial command 'z', see command *2.5.31 Setting the* reverse clearance.

SetNewMotorStepAngel

Definition:

bool **SetNewMotorStepAngele(***int stepAngle***)**

This function sets the motor step angle:

- *stepAngle* = 9 corresponds to 0.9°
- *stepAngle* = 18 corresponds to 1.8°
- *stepAngle* = 375 corresponds to 3.75°
- *stepAngle* = 75 corresponds to 75°
- *stepAngle* = 150 corresponds to 150°
- *stepAngle* = 180 corresponds to 180°

The value returned by the function can be used to check that the command was correctly recognized by the controller.

The function corresponds to the serial command 'a', see command *2.5.8 Setting the* step angle.

GetNewMotorStepAngel

Definition:

int **GetNewMotorStepAngel()**

This function reads the motor step angle:

- Return = 9 corresponds to 0.9°
- Return = 18 corresponds to 1.8°
- Return = 375 corresponds to 3.75°
- Return = 75 corresponds to 75°
- Return = 150 corresponds to 150°
- Return = 180 corresponds to 180°

The function corresponds to the serial command 'a', see command *2.5.8 Setting the* step angle.

SetNewAnalogueMin

Definition:

bool **SetNewAnalogueMin(***double analoqueMin***)**

This function sets the minimum analog voltage.

The value returned by the function can be used to check that the command was correctly recognized by the controller.

The function corresponds to the serial command 'Q', see command *2.7.3 Setting the minimum voltage* for the analog mode.

GetNewAnalogueMin

Definition:

double **GetNewAnalogueMin()**

This function reads the minimum analog voltage.

The function corresponds to the serial command 'Q', see command *2.7.3 Setting the minimum voltage* for the analog mode.

SetNewAnalogueMax

Definition:

bool **SetNewAnalogueMax(***double analoqueMin***)**

This function sets the maximum analog voltage.

The value returned by the function can be used to check that the command was correctly recognized by the controller.

The function corresponds to the serial command 'R', see command *2.7.4 Setting the maximum voltage* for the analog mode.

GetNewAnalogueMax

Definition:

double **GetNewAnalogueMax()**

This function reads the maximum analog voltage.

The function corresponds to the serial command 'R', see command *2.7.4 Setting the maximum voltage* for the analog mode.

SetNewAngelDeviationMax

Definition:

bool ***SetNewAngelDeviationMax(****int deviation***)**

Function sets the maximum angular deviation of the motor.

The value returned by the function can be used to check that the command was correctly recognized by the controller.

The function corresponds to the serial command 'X', see command *2.5.13 Setting the maximum encoder* deviation.

GetNewAngelDeviationMax

Definition:

double **GetNewAngelDeviationMax()**

This function reads the maximum angular deviation of the motor.

The function corresponds to the serial command 'X', see command *2.5.13 Setting the maximum encoder* deviation.

SetNewPositionType

Definition:

bool **SetNewPositionType(***int positionType***)**

This function sets the motor step angle:

- *positionType* = 1 corresponds to relative; depends on the operation mode
- *positionType* = 2 corresponds to absolute; depends on the operation mode
- *positionType* = 3 corresponds to internal reference run
- *positionType* = 4 corresponds to external reference run

The value returned by the function can be used to check that the command was correctly recognized by the controller.

The function corresponds to the serial command 'p', see commande *2.6.6 Setting positioning mode (old* scheme) and *2.6.7 Setting the positioning mode* (new scheme).

GetNewPositionType

Definition:

int **GetNewPositionType(***int operationNumber***)**

This function reads the motor step angle:

- Return = 1 corresponds to relative; depends on the operation mode
- Return = 2 corresponds to absolute; depends on the operation mode
- Return = 3 corresponds to internal reference run
- Return = 4 corresponds to external reference run

Here the *operationNumber* parameter is the record number (travel profile) from which the position type should be read.

The function corresponds to the serial command 'p', see command *2.6.6 Setting positioning mode (old* scheme) and *2.6.7 Setting the positioning mode* (new scheme).

SetNewSteps

Definition:

bool **SetNewSteps(***int steps***)**

This function sets the number of steps.

The value returned by the function can be used to check that the command was correctly recognized by the controller.

The function corresponds to the serial command 's', see command *2.6.8 Setting the* travel distance.

GetNewSteps

Definition:

int **GetNewSteps(***int operationNumber***)**

This function reads the number of steps.

Here the *operationNumber* parameter is the record number (travel profile) from which the position type should be read.

The function corresponds to the serial command 's', see command *2.6.8 Setting the* travel distance.

SetNewStartFrequency

Definition:

bool **SetNewStartFrequency(***int startFreuqenz***)**

This function sets the start frequency of the motor.

The value returned by the function can be used to check that the command was correctly recognized by the controller.

The function corresponds to the serial command 'u', see command *2.6.9 Setting the* minimum frequency.

GetNewStartFrequency

Definition:

int **GetNewStartFrequency(***int operationNumber***)**

This function reads the start frequency of the motor.

Here the *operationNumber* parameter is the record number (travel profile) from which the position type should be read.

The function corresponds to the serial command 'u', see command *2.6.9 Setting the* minimum frequency.

SetNewMaxFrequency

Definition:

bool **SetNewMaxFrequency(***int maxFreuqenz***)**

This function sets the target frequency of the motor.

The value returned by the function can be used to check that the command was correctly recognized by the controller.

The function corresponds to the serial command 'o', see command *2.6.10 Setting the* maximum frequency.

GetNewMaxFrequency

Definition:

int **GetNewMaxFrequency(***int operationNumber***)**

This function reads the target frequency of the motor.

Here the *operationNumber* parameter is the record number (travel profile) from which the position type should be read.

The function corresponds to the serial command 'o', see command *2.6.10 Setting the* maximum frequency.

SetNewMaxFrequency2

Definition:

bool **SetNewMaxFrequency2(***int maxFreuqenz***)**

This function sets the maximum frequency of the motor.

The value returned by the function can be used to check that the command was correctly recognized by the controller.

The function corresponds to the serial command 'n', see command *2.6.11 Setting the maximum* frequency 2.

GetNewMaxFrequency2

Definition:

int **GetNewMaxFrequency2(***int operationNumber***)**

This function reads the maximum frequency of the motor.

Here the *operationNumber* parameter is the record number (travel profile) from which the position type should be read.

The function corresponds to the serial command 'n', see command *2.6.11 Setting the maximum* frequency 2.

NewSuppressResponse

Definition:

bool **NewSuppressResponse(***int suppress***)**

This function activates or deactivates the response suppression on sending:

- *suppress* = 0 corresponds to switched on
- *suppress* = 1 corresponds to switched off

The value returned by the function can be used to check that the command was correctly recognized by the controller.

The function corresponds to the serial command '|', see command *2.6.4 Reading out the* current record.

SetNewRamp

Definition:

bool **SetNewRamp(***int ramp***)**

This function sets the ramp.

The value returned by the function can be used to check that the command was correctly recognized by the controller.

The function corresponds to the serial command 'b', see command *2.6.12 Setting the* acceleration ramp.

GetNewRamp

Definition:

int **GetNewRamp(***int operationNumber***)**

This function reads the ramp.

Here the *operationNumber* parameter is the record number (travel profile) from which the position type should be read.

The function corresponds to the serial command 'b', see command *2.6.12 Setting the* acceleration ramp.

SetNewRotationMode

Definition:

bool **SetNewRotationMode(***int rotationMode***)**

This function sets the encoder monitoring mode:

- *rotationMode* = 0 corresponds to switched off
- *rotationMode* = 1 corresponds to checking at the end
- *rotationMode* = 2 corresponds to checking in between

The value returned by the function can be used to check that the command was correctly recognized by the controller.

The function corresponds to the serial command 'U', see command *2.5.9 Setting the* error correction mode.

GetNewRotationMode

Definition:

int **GetNewRotationMode()**

This function reads the encoder monitoring mode:

- Return = 0 corresponds to switched off
- Return = 1 corresponds to checking at the end
- Return = 2 corresponds to checking in between

The function corresponds to the serial command 'U', see command *2.5.9 Setting the* error correction mode.

SetNewDirection

Definition:

bool **SetNewDirection(***int direction***)**

This function sets the direction of rotation of the motor.

Possible parameters are:

- 0 corresponds to the direction of rotation, left
- 1 corresponds to the direction of rotation, right

The value returned by the function can be used to check that the command was correctly recognized by the controller.

The function corresponds to the serial command 'd', see command *2.6.15 Setting the* direction of rotation.

GetNewDirection

Definition:

int **GetNewDirection(***int operationNumber***)**

This function reads the direction of rotation of the motor.

Possible return values are:

- 0 corresponds to the direction of rotation, left
- 1 corresponds to the direction of rotation, right

Here the *operationNumber* parameter is the record number (travel profile) from which the position type should be read.

The function corresponds to the serial command 'd', see command *2.6.15 Setting the* direction of rotation.

SetNewDirectionReverse

Definition:

bool **SetNewDirectionReverse(***bool directionReverse***)**

This function sets the reversal in the direction of rotation of the motor.

The value returned by the function can be used to check that the command was correctly recognized by the controller.

The function corresponds to the serial command 't', see command *2.6.16 Setting the* change of direction.

GetNewDirectionReverse

Definition:

bool **GetNewDirectionReverse(***int operationNumber***)**

This function reads the reversal in the direction of rotation of the motor.

Here the *operationNumber* parameter is the record number (travel profile) from which the position type should be read.

The function corresponds to the serial command 't', see command *2.6.16 Setting the* change of direction.

SetNewEncoderDirection

Definition:

bool **SetNewEncoderDirection(***bool encoderDirection***)**

This function sets whether the encoder rotation direction of the motor should be reversed.

The value returned by the function can be used to check that the command was correctly recognized by the controller.

The function corresponds to the serial command 'q', see command *2.5.11 Setting the* encoder direction.

GetNewEncoderDirection

Definition:

bool **GetNewEncoderDirection()**

This function reads whether the encoder rotation direction of the motor will be reversed.

The function corresponds to the serial command 'q', see command *2.5.11 Setting the* encoder direction.

SetNewBreak

Definition:

bool **SetNewBreak(***double breakTime***)**

This function sets the pause time of the motor.

The value returned by the function can be used to check that the command was correctly recognized by the controller.

The function corresponds to the serial command 'P', see command *2.6.18 Setting the* record pause.

GetNewBreak

Definition:

int **GetNewBreak(***int operationNumber***)**

This function reads the pause time of the motor.

Here the *operationNumber* parameter is the record number (travel profile) from which the position type should be read.

The function corresponds to the serial command 'P', see command *2.6.18 Setting the* record pause.

SetNewRepeat

Definition:

bool **SetNewRepeat(***int repeat***)**

This function sets the repetitions of the motor.

The value returned by the function can be used to check that the command was correctly recognized by the controller.

The function corresponds to the serial command 'W', see command *2.6.17 Setting the* repetitions.

GetNewRepeat

Definition:

int **GetNewRepeat(***int operationNumber***)**

This function reads the repetitions of the motor.

Here the *operationNumber* parameter is the record number (travel profile) from which the position type should be read.

The function corresponds to the serial command 'W', see command *2.6.17 Setting the* repetitions.

SetIO

Definition:

bool **SetIO(***int io***)**

This function sets the status of the inputs as integer mask.

The value returned by the function can be used to check that the command was correctly recognized by the controller.

The function corresponds to the serial command 'Y', see command *2.5.27 Setting the* outputs.

GetIO

Definition:

int **GetIO()**

This function reads the status of the inputs as integer mask.

The function corresponds to the serial command 'Y', see command *2.5.27 Setting the* outputs.

SetInputMask

Definition:

bool **SetInputMask(***int ioMask***)**

This function sets the mask of how the inputs should react (falling or rising flanks).

The value returned by the function can be used to check that the command was correctly recognized by the controller.

The function corresponds to the serial command 'L', see command *2.5.24 Masking and demasking the* inputs.

GetInputMask

Definition:

int **GetInputMask()**

This function reads the mask of how the inputs react (falling or rising flanks).

The function corresponds to the serial command 'L', see command *2.5.24 Masking and demasking the* inputs.

SetInputMaskEdge

Definition:

bool **SetInputMaskEdge(***int ioMask***)**

This function sets the mask of how the inputs should react (falling or rising flanks).

The value returned by the function can be used to check that the command was correctly recognized by the controller.

The function corresponds to the serial command 'h', see command *2.5.25 Reversing the polarity of the inputs* and outputs.

GetInputMaskEdge

Definition:

int **GetInputMaskEdge()**

This function reads the mask of how the inputs react (falling or rising flanks).

The function corresponds to the serial command 'h', see command *2.5.25 Reversing the polarity of the inputs* and outputs.

SetRampType

Definition:

bool **SetRampType(***int rampType***)**

This function sets the ramp type:

- *rampType* = 0 corresponds to trapezoidal ramp
- *rampType* = 1 corresponds to sinusoidal ramp

The value returned by the function can be used to check that the command was correctly recognized by the controller.

The function corresponds to the serial command 'ramp_mode', see command *2.5.32 Setting the* ramp.

GetRampType

Definition:

int **GetRampType()**

This function reads the ramp type:

- Return = 0 corresponds to trapezoidal ramp
- Return = 1 corresponds to sinusoidal ramp

The function corresponds to the serial command 'ramp_mode', see command *2.5.32 Setting the* ramp.

GetEncoderRotary

Definition:

int **GetEncoderRotary()**

This function reads the encoder position.

The function corresponds to the serial command 'I', see command *2.5.16 Reading out* the encoder position.

SetBraleTA

Definition:

bool **SetBraleTA(***uint32 brake***)**

This function sets the TA value for the external brake.

The value returned by the function can be used to check that the command was correctly recognized by the controller.

The function corresponds to the serial command 'brake_ta', see command *2.5.35 Setting the wait time for switching* off the brake voltage.

GetBrakeTA

Definition:

unit32 **GetBrakeTA()**

This function reads the TA value of the external brake.

The function corresponds to the serial command 'brake_ta', see command *2.5.35 Setting the wait time for switching* off the brake voltage.

SetBrakeTB

Definition:

bool **SetBrakeTB(***uint32 brake***)**

This function sets the TB value for the external brake.

The value returned by the function can be used to check that the command was correctly recognized by the controller.

The function corresponds to the serial command 'brake_tb', see command *2.5.36 Setting the wait time* for the motor movement.

GetBrakeTB

Definition:

unit32 **GetBrakeTB()**

This function reads the TB value of the external brake.

The function corresponds to the serial command 'brake_tb', see command *2.5.36 Setting the wait time* for the motor movement.

SetBrakeTC

Definition:

bool **SetBrakeTC(***uint32 brake***)**

This function sets the TC value for the external brake.

The value returned by the function can be used to check that the command was correctly recognized by the controller.

The function corresponds to the serial command 'brake_tc', see command *2.5.37 Setting the wait time for* switching off the motor current.

GetBrakeTC

Definition:

unit32 **GetBrakeTC()**

This function reads the TC value of the external brake.

The function corresponds to the serial command 'brake_tc', see command *2.5.37 Setting the wait time for* switching off the motor current.

SetRotencInc

Definition:

bool **SetRotencInc(**int *rotationInc***)**

This function sets the encoder resolution.

The value returned by the function can be used to check that the command was correctly recognized by the controller.

The function corresponds to the serial command 'CL_rotenc_inc ', see command *2.9.10 Setting the number of* increments.

GetRotencInc

Definition:

int **GetRotencInc()**

This function reads the encoder resolution.

The function corresponds to the serial command 'CL_rotenc_inc', see command *2.9.10 Setting the number of* increments.

4.4 Programming examples

Introduction

A brief list of example programs follows. The programs themselves are available as source code and in the compiled form in the “Example” directory of the NanoPro software. There is also a project file for Visual Studio for the examples.

OfficeExample

This example can be executed in Excel by means of VBA.

To carry out the example, the COM port and the transfer rate must be set.

There are 2 ways of actuating a motor:

- to let a number of steps be travelled
- to let a predefined record be carried out in the controller.

VBAExample

There are two modes in this example:

- the positioning mode and
- the speed mode.

The baud rate and the serial port must also be set. This example uses the PD4I.dll command as a reference in the project. No COM interface is used here.

The motor address 1 is assumed in this example.

VBACommandsPD4IExample-2Motor

This example is similar to the example of VBAExample, but two motors are actuated here.

ManagedC++CommandsPD4I

This is set up in exactly the same way as VBAExample, but Managed C++ Code is used here.

The motor address 1 is assumed in this example.

UnManagedC++CommandsPD4I

This is set up in exactly the same way as VBAExample, but Unmanaged C++ Code is used here.

The motor address 1 is assumed in this example.

Index

A

C

D

E

F

I

J

K

L

M

N

P

R

S

T