Programming manual for stepper motor positioning controls

Valid from firmware 23 Aug 2010

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.de

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.de

Version/Change overview

Version	Date	Changes
V1.0	02/10/2009	Command reference created (firmware version 04.12.2008)
V2.0	12/11/2009	Instruction sets (firmware version 10.10.2009), supplemented by Java programming and COM interface programming, hence renamed as “Programming Manual”
V2.1	01/28/2010	Instruction sets
V2.2	02/11/2010	Jerkfree ramp instruction set
V2.3	04/08/2010	New notice: Java programs and serial communication possible at the same time
V2.4	11/03/2010	Instruction sets and corrections for serial communication and Java programming; Revision of COM interface programming

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
- SMCI33 *
- SMCI35
- SMCI36
- SMCI47-S *
- SMCP33
- PD4-N
- PD6-N

* Please note following information!

Information on SMCI33 and SMCI47-S

For drivers with firmware older than 30 April 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.de

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 `'+'` and `'-'`.

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.

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

Example

Value transmitted to the controller:	`'#1G10000000\r'`
Firmware response:	`'1G10000000\r'` (value is ignored)

Examples

Setting the travel distance of controller 1:	`'#1s1000\r'` → `'1s1000\r'`
Starting a record:	`'#1A\r'` → `'1A\r'`
Invalid command:	`'#1°\r'` → `'1°?\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_gibt_es_nicht\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>:Schlüsselwort_Kommando_abc\r'`

Firmware response

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

Response: `'<ID>:Schlüsselwort_Kommando_abc+Wert\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>:Schlüsselwort_Kommando_abc=Wert\r'`

Firmware response

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

Response: `'<ID>:Schlüsselwort_Kommando_abc=Wert\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_pp=100\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

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

Example

Reading out the travel distance: `'#1Zs\r'` → `'1Zs1000\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: `'#1Z5s'` → `'1Z5s2000'`

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 the positioning mode* (new scheme)	46
Travel distance	's'	*2.6.7 Setting the* travel distance	48
Initial step frequency	'u'	*2.6.8 Setting the* minimum frequency	48
Maximum step frequency	'o'	*2.6.9 Setting the* maximum frequency	49
Second maximum step frequency	'n'	*2.6.10 Setting the maximum* frequency 2	49
Acceleration ramp	'b'	*2.6.11 Setting the* acceleration ramp	50
Brake ramp	'B'	2.6.12 *Setting the* brake ramp	50
Maximum jerk for acceleration ramp	':b'	*2.5.36 Setting the maximum jerk for* the acceleration ramp	37
Maximum jerk for brake ramp	':B'	*2.5.37 Setting the maximum jerk for* the braking ramp	37
Direction of rotation	'd'	*2.6.14 Setting the* direction of rotation	51
Reversal of direction of rotation for repeat records	't'	*2.6.15 Setting the* change of direction	52
Repetitions	'W'	*2.6.16 Setting the* repetitions	52
Pause between repetitions and continuation records	'P'	*2.6.17 Setting the* record pause	53
Record number of continuation record	'N'	*2.6.18 Setting the* continuation record	53

2.5 General commands

2.5.1 Setting the motor type

Parameter

Symbol	Permissible values	Writable	Data type	Default value
`':CL_motor_type'`	0 to 2	Yes	u16 (integer)	0

Firmware response

Confirms the command through an echo.

Description

Defines the connected motor type:

- Value 0: stepper motor
- Value 1: BLDC motor with hall sensors
- Value 2: BLDC motor with hall sensors and encoder

Reading out

Command `':CL_motor_type'` is used to read out the current setting of the value.

2.5.2 Setting the phase current

Parameter

Symbol	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.

2.5.3 Setting the phase current at standstill

Parameter

Symbol	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.

2.5.4 Setting the peak current for BLDC

Parameter

Symbol	Permissible values	Writable	Data type	Default value
':ipeak'	0 to 150	Yes	u8 (integer)	0

Firmware response

Confirms the command through an echo.

Description

Sets the peak current for BLDC motors in percent. This value must be at least as large as the set phase current; otherwise, the phase current value is used.

Reading out

Command `':ipeak'` is used to read out the current setting of the value.

2.5.5 Setting the current time constant for BLDC

Parameter

Symbol	Permissible values	Writable	Data type	Default value
`':itime'`	0 to 65535	Yes	u16 (integer)	0

Firmware response

Confirms the command through an echo.

Description

Sets the current time constant for BLDC motors in ms. This defines the duration for which the set peak current can flow.

Reading out

Command `':itime'` is used to read out the current setting of the value.

2.5.6 Setting the step mode

Parameter

Symbol	Permissible values	Writable	Data type	Default value
`'g'`	1, 2, 4, 5, 8, 10, 16, 32, 64, 254, 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 254 which selects the feed rate mode, and with the exception of the value 255 which selects the adaptive step mode.

Feed rate mode is contained in firmware later than 15 March 2010.

Reading out

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

2.5.7 Setting the drive address

Parameter

Symbol	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 only one controller is connected and that the newly set address is not already occupied by another motor as this would make communication impossible.

Also, if there is an address rotary switch on the controller, it must be set to 0 since otherwise the address set by the switch is used.

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

2.5.8 Setting the motor ID

Parameter

Symbol	Permissible values	Writable	Data type	Default value
`':mt'`	0 to 2147483647	Yes	u32	0

Firmware response

Confirms the command through an echo.

Description

Returns and sets the motor ID set in NanoPro.

This motor ID uniquely identifies the motor type, motor designation and connection type (e.g. ST5918 connected in parallel) and is used to store in the controller which motor is currently connected (used by NanoPro to determine the maximum permissible phase current, for example).

Reading out

Command `':mt'` is used to read out the current setting of the value.

2.5.9 Setting the limit switch behavior

Parameter

Symbol	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 forwards
Bit10: Free 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.

2.5.10 Setting the error correction mode

Parameter

Symbol	Permissible values	Writable	Data type	Default value
`'U'`	0 to 2	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
- Value 2: Correction during 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.

The "Correction during travel" setting exists for compatibility reasons and is equivalent to the "Correction after travel" behavior. To actually make a correction during travel, the closed loop mode should be used.

Reading out

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

2.5.11 Setting the record for auto correction

Parameter

Symbol	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.

If 0 is set, no correction run is performed; instead, an error is output if the error correction (command `'U'`) is activated.

See command *2.5.10 Setting the* error correction mode `'U'`.

Reading out

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

2.5.12 Setting the encoder direction

Parameter

Symbol	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.

2.5.13 Setting the swing out time

Parameter

Symbol	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 *2.5.10 Setting the* error correction mode `'U'`.

Between repetitions or continuation records, this position is only checked if the pause time (see command *2.6.17 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.

2.5.14 Setting the maximum encoder deviation

Parameter

Symbol	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.

2.5.15 Setting the feed rate numerator

Parameter

Symbol	Permissible values	Writable	Data type	Default value
`':feed_const_num'`	0 to 2147483647	Yes	u32 (integer)	0

Firmware response

Confirms the command through an echo.

Description

Sets the numerator for the feed rate. This value defines the number of steps per rotation of the motor shaft for the feed rate step mode. The feed rate is only used if numerator and the denominator are not equal to 0. Otherwise, the encoder resolution is used.

Reading out

Command `':feed_const_num'` is used to read out the current setting of the value.

2.5.16 Setting the feed rate denominator

Parameter

Symbol	Permissible values	Writable	Data type	Default value
`':feed_const_denum'`	0 to 2147483647	Yes	u32 (integer)	0

Firmware response

Confirms the command through an echo.

Description

Sets the denominator for the feed rate. This value defines the number of steps per rotation of the motor shaft for the feed rate step mode. The feed rate is only used if numerator and the denominator are not equal to 0. Otherwise, the encoder resolution is used.

Reading out

Command `':feed_const_denum'` is used to read out the current setting of the value.

2.5.17 Resetting the position error

Parameter

Symbol	Permissible values	Writable	Data type	Default value
`'D'`	-100000000 to +100000000	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 *2.5.18* and *2.5.19*).

For input with parameters, C and I are set to the parameter value.
Ex.: `'D100'` → C=100; I=100

2.5.18 Reading out the error memory

Parameter

Symbol	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 *2.5.22 Reading out* the status `'$'` command.

2.5.19 Reading out the encoder position

Parameter

Symbol	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 *2.5.20 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.

2.5.20 Reading out the position

Parameter

Symbol	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 ± 100000000).

2.5.21 Request “Motor is referenced”

Parameter

Symbol	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 *2.5.17 Resetting the* position error.

2.5.22 Reading out the status

Parameter

Symbol	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 and 6 are always set to 1, bits 5 and 7 are always set to 0.

2.5.23 Reading out the firmware version

Parameter

Symbol	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: possible SMCI47-S, PD6-N, PD4-N, SMCI33, SMCI35, SMCI36, SMCI12, SMCP33

`'_'` 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'`

2.5.24 Reading out the operating time since the firmware update

Parameter

Symbol	Permissible values	Writable	Data type	Default value
`':optime'`	–	No	–	–

Firmware response

Returns the operating time of the controller.

Description

Delivers the operating time of the controller since the last firmware update in seconds. When a firmware update is performed, the value is reset to 0 and counting starts from the beginning.

2.5.25 Setting the function of the digital inputs

Parameter

Symbol	Permissible values	Writable	Data type	Default value
`':port_in_a'` to `':port_in_h'`	0 to 13	Yes	u8 (integer)	Different for each input

Firmware response

Confirms the command through an echo.

Description

Sets the function of each digital input. Each function is represented by a unique number:

Input function	Number
User defined	0
Start record/error reset	1
Record selection bit 0	2
Record selection bit 1	3
Record selection bit 2	4
Record selection bit 3	5
Record selection bit 4	6
External reference switch	7
Trigger	8
Direction	9
Enable	10
Clock	11
Clock direction mode, mode selection 1	12
Clock direction mode, mode selection 2	13

User-defined (0) means that the input/output is not used by the firmware and is available to the user as a general purpose I/O.

Examples

- Defining input 3 as a trigger input for controller 1: `'#1:port_in_c8\r'`
- Defining input 6 as a clock input for controller 2: `'#2:port_in_f11\r'`

Reading out

The commands `':port_in_a'` to `':port_in_h'` without an argument can be used to read out the function set for the respective input.

2.5.26 Setting the function of the digital outputs

Parameter

Symbol	Permissible values	Writable	Data type	Default value
':port_out_a' to ':port_out_h'	0 to 2	Yes	u8 (integer)	Different for each output

Firmware response

Confirms the command through an echo.

Description

Sets the function of each digital output. Each function is represented by a unique number:

Input function	Number
User defined	0
Ready	1
Running	2

User-defined (0) means that the input/output is not used by the firmware and is available to the user as a general purpose I/O.

Examples

- Defining output 1 for the travel display for controller 1: `'#1:port_out_a2\r'`
- Defining output 2 for the ready display for controller 2: `'#2:port_out_b1\r'`

Reading out

The commands `':port_out_a'` to `':port_out_h'` without an argument can be used to read out the function set for the respective output.

2.5.27 Reversing the polarity of the inputs and outputs

Parameter

Symbol	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.

2.5.28 Setting the debounce time for the inputs

Parameter

Symbol	Permissible values	Writable	Data type	Default value
'K'	0 to 250	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.

2.5.29 Setting the outputs

Parameter

Symbol	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 **Fehler! Verweisquelle konnte nicht gefunden werden. Fehler! Verweisquelle konnte nicht gefunden werden.** `'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'.

2.5.30 Reading out EEPROM byte (read EE byte)

Parameter

Symbol	Permissible values	Writable	Data type	Default value
'(E'	0 to 65535	No	u16	0

Firmware response

Returns the value of the byte in the EEPROM at the address passed in the parameter.

Description

Reads a byte out of the EEPROM and returns the value of this byte.

2.5.31 Carrying out an EEPROM reset

Parameter

Symbol	Permissible values	Writable	Data type	Default value
'~'	–	No	–	–

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.

A motor should not be connected during a reset. After the reset, the controller should be disconnected from the power supply for a few seconds.

2.5.32 Setting automatic sending of the status

Parameter

Symbol	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 *2.5.22 Reading out* the status '$', with the difference that a lower case '$' is sent instead of the 'j'.

Reading out

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

2.5.33 Starting the bootloader

Parameter

Symbol	Permissible values	Writable	Data type	Default value
'@S'	–	No	–	–

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.

2.5.34 Setting the reverse clearance

Parameter

Symbol	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.

2.5.35 Setting the ramp

Parameter

Symbol	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 `'= + Wert'`, the current setting of the value can be read out.

2.5.36 Setting the maximum jerk for the acceleration ramp

Parameter

Symbol	Permissible values	Writable	Data type	Default value
':b'	1 to 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 `'= + Wert'`, 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)

2.5.37 Setting the maximum jerk for the braking ramp

Parameter

Symbol	Permissible values	Writable	Data type	Default value
':B'	1 to 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 `'= + Wert'`, 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)

2.5.38 Setting the wait time for switching off the brake voltage

Parameter

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

Unit

ms

Firmware response

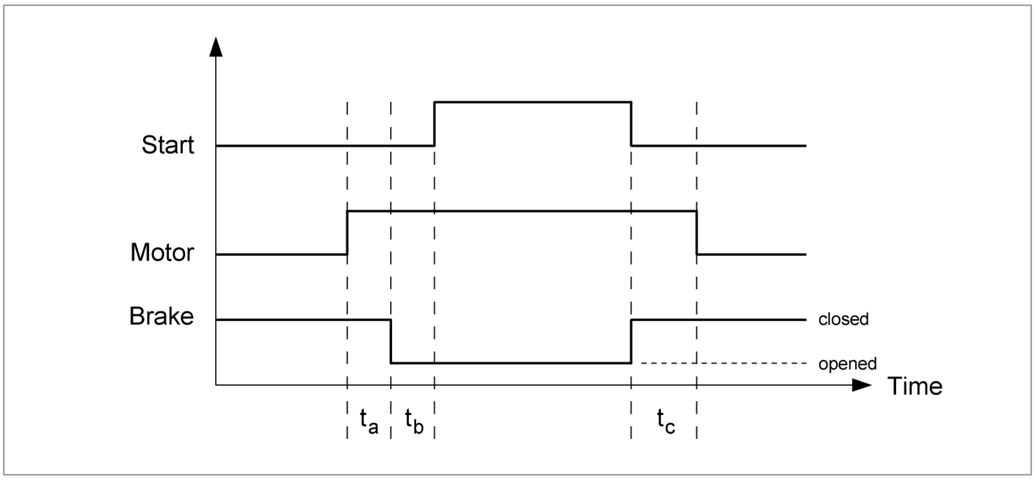
Confirms the command through an echo.

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. 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 motor current is switched off by resetting the enable input (see Section *2.5.25* "*Setting the function of* the digital inputs").

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 `'= + Wert'`, the current setting of the value can be read out.

2.5.39 Setting the wait time for the motor movement

Parameter

Symbol	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 *2.5.38 Setting the wait time for switching off the brake voltage* `'ta'`.

Reading out

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

2.5.40 Setting the wait time for switching off the motor current

Parameter

Symbol	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 *2.5.38 Setting the wait time for switching off the brake voltage* `'ta'`.

Reading out

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

2.5.41 Setting baudrate of the controller

Parameter

Symbol	Permissible values	Writable	Data type	Default value
`':baud'`	1 to 12	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 baud rate of the 1st controller to 14400 baud.

Reading out

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

2.5.42 Setting the CRC checksum

Parameter

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

Firmware response

Confirms the command through an echo.

Description

Switches the check of the serial communication by means of a CRC checksum (*cyclic redundancy check*) on or off:

- Value 0: CRC check deactivated
- Value 1: CRC check activated

Attention:

For communication with the controller after the CRC check is activated, the correct CRC checksum must be included with each command, separated from the command by a tab. If not, the controller does not execute the command and sends the response `'<Befehl>?crc<Tab><Prüfsumme>'`.

Reading out

Command `':crc'` is used to read out the current setting of the value.

2.5.43 Setting the correction of the sinus commutation

Parameter

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

Firmware response

Confirms the command through an echo.

Description

Switches the correction of the sinus commutation on or off:

- Value 0: correction deactivated
- Value 1: correction activated

Note:
This function only has an affect on calibrated motors.

Reading out

Command `':cal_elangle_enable'` is used to read out the current setting of the value.

2.5.44 Setting the electrical angle

Parameter

Symbol	Permissible values	Writable	Data type	Default value
`':cal_elangle_data'`	0 to 2147483647	No	s32	0

Firmware response

Confirms the command through an echo.

Description

Sets the value of the electrical angle.

Reading out

Command `':cal_elangle_data'` is used to read out the current value.

2.5.45 Hall configuration

Parameter

Symbol	Permissible values	Writable	Data type	Default value
`':hall_mode'`	0 to 16777215	Yes	u32	2371605

Firmware response

Confirms the command through an echo.

Description

The hall mode specifies the hall configuration of a connected brushless motor as an integer value. For example, motor types DB42S03, DB22M and DB87S01 require the value 2371605 (0x243015 hexadecimal) and motor types DB57 and DB22L require value 5309250 (0x510342 hexadecimal).

The correct value can conveniently be set via NanoPro for all Nanotec motors.

Reading out

Command `':hall_mode'` is used to read out the current setting of the value.

2.6 Record commands

2.6.1 Starting the motor

Parameter

Symbol	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

Parameter

Symbol	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

Symbol	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

Parameter

Symbol	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

Symbol	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 the positioning mode* (new scheme)	46
Travel distance	's'	*2.6.7 Setting the* travel distance	48
Initial step frequency	'u'	*2.6.8 Setting the* minimum frequency	48
Maximum step frequency	'o'	*2.6.9 Setting the* maximum frequency	49
Second maximum step frequency	'n'	*2.6.10 Setting the maximum* frequency 2	49
Acceleration ramp	'b'	*2.6.11 Setting the* acceleration ramp	50
Brake ramp	'B'	2.6.12 *Setting the* brake ramp	50
Maximum jerk for acceleration ramp	':b'	*2.5.36 Setting the maximum jerk for* the acceleration ramp	37
Maximum jerk for brake ramp	':B'	*2.5.37 Setting the maximum jerk for* the braking ramp	37
Direction of rotation	'd'	*2.6.14 Setting the* direction of rotation	51
Reversal of direction of rotation for repeat records	't'	*2.6.15 Setting the* change of direction	52
Repetitions	'W'	*2.6.16 Setting the* repetitions	52
Pause between repetitions and continuation records	'P'	*2.6.17 Setting the* record pause	53
Record number of continuation record	'N'	*2.6.18 Setting the* continuation record	53

2.6.6 Setting the positioning mode (new scheme)

Parameter

Symbol	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, the positioning mode `'p'` is set to 1.

Description

The positioning modes `'p'` are:

Positioning mode	
p=1	Relative positioning; The command *2.6.7 Setting the* travel distance `'s'` specifies the travel distance relative to the current position. The command *2.6.14 Setting the* direction of rotation `'d'` specifies the direction. The parameter *2.6.7 Setting the* travel distance `'s'` must be positive.
p=2	Absolute positioning; Command *2.6.7 Setting the* travel distance `'s'` defines the target position relative to the reference position. Command *2.6.14 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 *2.6.14 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 *2.5.9 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 *2.6.14 Setting the* direction of rotation `'d'` until it reaches the limit switch. Then a free run is performed, depending on the setting. See command *2.5.9 Setting the* limit switch behavior `'l'`.
Speed mode	
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	
p=6	Flag positioning mode; After starting, the motor runs up to the maximum speed. After arrival of the trigger event (command *2.7.12 Actuating the* trigger `'T'` or trigger input) the motor continues to travel the selected travel distance (command *2.6.7 Setting the* travel distance `'s'`) and changes its speed to the maximum speed 2 (command *2.6.10 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	
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	
p=11	Analog mode
Joystick mode	
p=12	Joystick mode
Analog positioning mode	
p=13	Analog positioning mode
p=3	Internal reference run; see Positioning mode
p=4	External reference run; see Positioning mode
HW reference mode	
p=14	HW reference mode
Torque mode	
p=15	Torque mode
CL quick test mode	
p=16	CL quick test mode
CL test mode	
p=17	CL test mode

Reading out

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

2.6.7 Setting the travel distance

Parameter

Symbol	Permissible values	Writable	Data type	Default value
's'	-100000000 to +100000000	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 *2.6.14 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 *2.6.14 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.

2.6.8 Setting the minimum frequency

Parameter

Symbol	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 *2.6.11 Setting the* acceleration ramp `'b'`) to the maximum speed (command *2.6.9 Setting the* maximum frequency `'o'`).

Reading out

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

2.6.9 Setting the maximum frequency

Parameter

Symbol	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.

2.6.10 Setting the maximum frequency 2

Parameter

Symbol	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 the positioning mode* (new scheme).

Reading out

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

2.6.11 Setting the acceleration ramp

Parameter

Symbol	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.

2.6.12 Setting the brake ramp

Parameter

Symbol	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. The value 0 means that the value set for the acceleration ramp is used for the brake ramp.

Reading out

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

2.6.13 Setting the quickstop ramp

Parameter

Symbol	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.
Travel is stopped abruptly at a value of 0.

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

Reading out

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

2.6.14 Setting the direction of rotation

Parameter

Symbol	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.

2.6.15 Setting the change of direction

Parameter

Symbol	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 *2.6.16 Setting the* repetitions `'W'`.

Reading out

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

2.6.16 Setting the repetitions

Parameter

Symbol	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.

2.6.17 Setting the record pause

Parameter

Symbol	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.

2.6.18 Setting the continuation record

Parameter

Symbol	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

Symbol	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

Symbol	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. The software filter can be configured with the `'f'` command. Two different filter functions are available according to the value passed with the parameter:

- 0 – 16: simple average of the number of samples

Value for "f" command	Average value of ... values (1kHz sample rate)
0	1
1	1
2	2
3	3
4	4
5	5
6	6
7	7

Value for "f" command	Average value of ... values (1kHz sample rate)
8	8
9	9
10	10
11	11
12	12
13	13
14	14
15	15
16	16

-
- 17 – 255: recursive filter with separately adjustable time constant (time after which the filter output has approached the filter input to within 50%) and hysteresis (maximum change of the value at the filter input toward which the filter output is insensitive);
  f = (bit 0-3: power of two of the time constant in ms; Bit 4-7: size of the hysteresis) + 16

		Hysteresis in bit (+-20mV)														
		0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**
T in ms to reach 50% of the final value	**0**	32	48	64	80	96	112	128	144	160	176	192	208	224	240	0
	1	17	33	49	65	81	97	113	129	145	161	177	193	209	225	241
	2	18	34	50	66	82	98	114	130	146	162	178	194	210	226	242
	4	19	35	51	67	83	99	115	131	147	163	179	195	211	227	243
	8	20	36	52	68	84	100	116	132	148	164	180	196	212	228	244
	16	21	37	53	69	85	101	117	133	149	165	181	197	213	229	245
	32	22	38	54	70	86	102	118	134	150	166	182	198	214	230	246
	64	23	39	55	71	87	103	119	135	151	167	183	199	215	231	247
	128	24	40	56	72	88	104	120	136	152	168	184	200	216	232	248
	256	25	41	57	73	89	105	121	137	153	169	185	201	217	233	249
	512	26	42	58	74	90	106	122	138	154	170	186	202	218	234	250
	1024	27	43	59	75	91	107	123	139	155	171	187	203	219	235	251
	2048	28	44	60	76	92	108	124	140	156	172	188	204	220	236	252
	4096	29	45	61	77	93	109	125	141	157	173	189	205	221	237	253
	8192	30	46	62	78	94	110	126	142	158	174	190	206	222	238	254
	16384	31	47	63	79	95	111	127	143	159	175	191	207	223	239	255

Reading out

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

2.7.3 Setting the minimum voltage for the analog mode

Parameter

Symbol	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

Symbol	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 Setting the offset of the analog input

Parameter

Symbol	Permissible values	Writable	Data type	Default value
':aoa'	32768 to 32767	Yes	s16	0

Firmware response

Confirms the command through an echo.

Description

Sets the offset of the analog input.

Reading out

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

2.7.6 Setting the gain of the analog input

Parameter

Symbol	Permissible values	Writable	Data type	Default value
':aaa'	0 to 65535	Yes	u16	32768

Firmware response

Confirms the command through an echo.

Description

Sets the gain of the analog input. A higher value results in a higher gradient of the correction curves.

Reading out

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

2.7.7 Resetting switch-on numerator

Parameter

Symbol	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.

2.7.8 Adjusting the time until the current reduction

Parameter

Symbol	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.

2.7.9 Increasing the speed

Parameter

Symbol	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.

2.7.10 Reducing the speed

Parameter

Symbol	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.

2.7.11 Reading out the speed

Parameter

Symbol	Permissible values	Writable	Data type	Default value
`':v'`	– 2147483648 to 2147483647	No	s32	0

Firmware response

Confirms the command through an echo.

Description

States the current motor speed (only in speed mode).

Reading out

Command `':v'` is used to read out the current value when closed loop mode is active.

2.7.12 Actuating the trigger

Parameter

Symbol	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.

2.7.13 Setting the interpolation time period for the clock direction mode

Parameter

Symbol	Permissible values	Writable	Data type	Default value
`':clock_interp'`	0 to 16383	Yes	u16 (integer)	320

Firmware response

Confirms the command through an echo.

Description

Sets the interpolation time period for the clock direction mode in 33 microsecond-steps.

Example

Set value: 320 – one clock signal at the clock input is processed within 320 * 33 µs =~ 10 ms.

Reading out

Command `':clock_interp'` is used to read out the current setting of the value.

2.8 Commands for JAVA program

2.8.1 Transferring a Java program to the controller

Parameter

Symbol	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

Parameter

Symbol	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

Parameter

Symbol	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 Automatically starting the Java program when switching on the controller

Parameter

Symbol	Permissible values	Writable	Data type	Default value
'(JB'	0 to 1	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

The function should only be selected if

- a Java program is present on the controller
- the program has already been tested and is OK
- no infinite loops with send commands occur in the program

Otherwise, this command causes an overflow at the interface when the controller restarts and the program can no longer be stopped.

2.8.5 Reading out error of the Java program

Parameter

Symbol	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.

2.8.6 Reading out the warning of the Java program

Parameter

Symbol	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

Parameter

Symbol	Permissible values	Writable	Data type	Default value
':CL_enable'	0 to 2	Yes	u8 (integer)	0

Firmware response

Confirms the command through an echo.

Description

If the value is set to `'1'` or `'2'`, the firmware is instructed to activate the closed loop. This requires that the index position of the encoder is reached once.

- With the setting `'1'`, the closed loop is activated when the index is detected and the controller is back in ready status ("Auto enable after travel").
- With the setting `'2'`, the closed loop is activated after the index is detected ("Auto enable during travel").

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 *2.9.10 Setting the motor pole* pairs, *2.9.11 Setting the number of* increments and *2.9.12 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 17: 'p17").

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 `'= + Wert'`, the current setting of the value can be read out.

2.9.2 Reading out the closed loop mode status

Parameter

Symbol	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 control type for the speed mode

Parameter

Symbol	Permissible values	Writable	Data type	Default value
`':speedmode_control'`	0 and 1	Yes	u8 (integer)	0

Firmware response

Confirms the command through an echo.

Description

Specifies the control type for the speed mode:

- `'0'` = velocity loop
- `'1'` = position loop

This parameter defines the type of control loop that is used for controlling in speed mode if the closed loop is activated.

Reading out

Command `':speedmode_control'` is used to read out the current setting of the value.

2.9.4 Setting the tolerance window for the limit position

Parameter

Symbol	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 *2.9.5 Setting the time for the tolerance* window of the limit position.

Reading out

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

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

Parameter

Symbol	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 *2.9.4. Setting the tolerance window* for the limit position.

Reading out

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

2.9.6 Setting the maximum allowed following error

Parameter

Symbol	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 *2.9.7 Setting the time for the* maximum following error.

Reading out

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

2.9.7 Setting the time for the maximum following error

Parameter

Symbol	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 *2.9.6 Setting the maximum allowed* following error.

Reading out

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

2.9.8 Maximum speed deviation

Parameter

Symbol	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'` parameter can be used to specify for how long the speed deviation may be greater than the tolerance. For more information, see command *2.9.9 Time for maximum speed deviation*.

Reading out

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

2.9.9 Time for maximum speed deviation

Parameter

Symbol	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 *2.9.8 Maximum speed deviation*.

Reading out

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

2.9.10 Setting the motor pole pairs

Parameter

Symbol	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 ¼ of the number of full steps per rotation for stepper motors and 1/6 of the number of full steps per rotation for BLDC motors. The usual values are currently 50 and 100 for stepper motors and 2 and 4 for BLDC motors. Incorrect values will result in the closed loop not functioning properly.

Reading out

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

2.9.11 Setting the number of increments

Parameter

Symbol	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 *2.9.12 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 `'= + Wert'`, the current setting of the value can be read out.

2.9.12 Setting the number of revolutions

Parameter

Symbol	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 *2.9.11 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 `'= + Wert'`, the current setting of the value can be read out.

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

Parameter

Symbol	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 `'= + Wert'`, the current setting of the value can be read out.

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

Parameter

Symbol	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 `'= + Wert'`, the current setting of the value can be read out.

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

Parameter

Symbol	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 `'= + Wert'`, the current setting of the value can be read out.

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

Parameter

Symbol	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 `'= + Wert'`, the current setting of the value can be read out.

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

Parameter

Symbol	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 `'= + Wert'`, the current setting of the value can be read out.

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

Parameter

Symbol	Permissible values	Writable	Data type	Default value
`':CL_KD_v_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 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 `'= + Wert'`, the current setting of the value can be read out.

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

Parameter

Symbol	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 `'= + Wert'`, the current setting of the value can be read out.

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

Parameter

Symbol	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 `'= + Wert'`, the current setting of the value can be read out.

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

Parameter

Symbol	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 `'= + Wert'`, the current setting of the value can be read out.

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

Parameter

Symbol	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 `'= + Wert'`, the current setting of the value can be read out.

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

Parameter

Symbol	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 `'= + Wert'`, the current setting of the value can be read out.

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

Parameter

Symbol	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 `'= + Wert'`, the current setting of the value can be read out.

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

Parameter

Symbol	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 `'= + Wert'`, the current setting of the value can be read out.

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

Parameter

Symbol	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 `'= + Wert'`, the current setting of the value can be read out.

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

Parameter

Symbol	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 `'= + Wert'`, the current setting of the value can be read out.

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

Parameter

Symbol	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 `'= + Wert'`, the current setting of the value can be read out.

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

Parameter

Symbol	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 `'= + Wert'`, the current setting of the value can be read out.

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

Parameter

Symbol	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 `'= + Wert'`, the current setting of the value can be read out.

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

Parameter

Symbol	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 `'= + Wert'`, the current setting of the value can be read out.

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

Parameter

Symbol	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 `'= + Wert'`, the current setting of the value can be read out.

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

Parameter

Symbol	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 `'= + Wert'`, the current setting of the value can be read out.

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

Parameter

Symbol	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 `'= + Wert'`, the current setting of the value can be read out.

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

Parameter

Symbol	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 `'= + Wert'`, the current setting of the value can be read out.

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

Parameter

Symbol	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 `'= + Wert'`, the current setting of the value can be read out.

2.9.37 Setting the sampling point spacing of the load angle curve

Parameter

Symbol	Permissible values	Writable	Data type	Default value
':CL_la_node_distance'	1 to 65535	Yes	u16 (integer)	4096

Firmware response

Confirms the command through an echo.

Description

Sets the sampling point spacing for the load angle curve.

Reading out

Command `':CL_la_node_distance'` is used to read out the current setting of the value.

2.9.38 Setting the lower limit for the cascade controller

Parameter

Symbol	Permissible values	Writable	Data type	Default value
':ca'	0 to 2147483647	Yes	u32	327680

Firmware response

Confirms the command through an echo.

Description

This command is used to set the lower limit above which the cascade controller should be connected. Thus, a hysteresis can be set together with the `':cs'` command.Reading out

Command `':ca'` is used to read out the current setting of the value.

2.9.39 Setting the upper limit for the cascade controller

Parameter

Symbol	Permissible values	Writable	Data type	Default value
':cs'	0 to 2147483647	Yes	u32	512

Firmware response

Confirms the command through an echo.

Description

This command is used to set the upper limit up which the cascade controller is connected. Thus, a hysteresis can be set together with the `':ca'` command.

Reading out

Command `':cs'` is used to read out the current setting of the value.

2.9.40 Reading out the status of the cascade controller

Parameter

Symbol	Permissible values	Writable	Data type	Default value
`':ce'`	0 and 1	No	u8	0

Firmware response

Confirms the command through an echo.

Description

Specifies whether the cascade controller is currently active.

Reading out

Command `':ce'` is used to read out the current value.

2.10 Motor-dependent load angle 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 load angle values are determined by the controller and stored.

These load angle 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

Parameter

Symbol	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 Setting/reading out load angle measurement values of the motor

Parameter

Symbol	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 velocity-dependent load angle measurement values (closed loop load angle) of the motor determined during the test run are read out with the following commands and can be set again with these commands:

- ':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'

Reading out

Command `':CL_la_a'` to `':CL_la_j'` is used to read out the current setting of the value.

2.10.3 Reading out the velocity measurement values of the test run

Parameter

Symbol	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 speed measurement values (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'`

These values can only be read out after the test run. They indicate the velocities at which the corresponding load angle was measured. They are not stored in the EEPROM and therefore disappear after the controller is restarted.

2.10.4 Reading out current measurement values of the test run

Parameter

Symbol	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 current measurement values (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'`

These values can only be read out after the test run. They specify the currents at which the load angle was measured. They are not stored in the EEPROM and therefore disappear after the controller is restarted.

2.10.5 Reading out load angle measurement values of the test run

Parameter

Symbol	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 load angle measurement values (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'`

These values can only be read out after the test run. They specify the measured load angles and are a copy of the CL_öa_* values. They are not stored in the EEPROM and therefore disappear after the controller is restarted.

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 if `':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

Parameter

Symbol	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.

`'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

Parameter

Symbol	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

`'1'` = the setpoint position is selected

`'0'` = the setpoint position is deselected

2.11.4 Reading out the actual position of the encoder

Parameter

Symbol	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

`'1'` = the actual position is selected

`'0'` = the actual position is deselected

2.11.5 Reading out the setpoint current of the motor controller

Parameter

Symbol	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

Parameter

Symbol	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

Parameter

Symbol	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

Parameter

Symbol	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

Parameter

Symbol	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 utilization of the CAN bus in %.

Example

`':Capt_iBus=1'` the utilization of the CAN bus is selected

`':Capt_iBus=0'` the utilization of the CAN bus is deselected

2.11.10 Reading out the controller temperature

Parameter

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

Priority

8

Unit

Value range 0 – 1023

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

Temperature curve

The controllers output the raw measurement value of the A/D converter. To calculate the temperature of the controller from this value, the temperature curve of the measurement sensor must be included in the calculation.

Conversion

The conversion of the raw material value x in the temperature T (°C) uses the following formula:

T = [1266500 / (4250 + log((x/1023) * 0.33 / (1-(x/1023))) * 298)] – 273

Value table

Measured value x	Temperature T (°C)	Measured value x	Temperature T (°C)
5	97.48	520	35.09
20	78.82	540	34.33
40	70.03	560	33.57
60	64.98	580	32.82
80	61.41	600	32.05
100	58.64	620	31.28
120	56.36	640	30.5
140	54.42	660	29.71
160	52.71	680	28.9
180	51.19	700	28.07
200	49.8	720	27.22
220	48.53	740	26.34
240	47.35	760	25.43
260	46.24	780	24.48
280	45.2	800	23.48
300	44.21	820	22.41
320	43.26	840	21.28
340	42.34	860	20.05
360	41.46	880	18.71
380	40.61	900	17.21
400	39.78	920	15.5
420	38.97	940	13.5
440	38.17	960	11.03
460	37.39	980	7.75
480	36.62	1000	2.64
500	35.85	1020	-12.45
		1022	-19.87

Programming example (C#)

```
double computeTemperature(UInt16 value) {
double adc_max = 1023;
double R0 = 33000;
double TnK = 298;
```

```
double BK = 4250;
double Rn = 100000;
double bruch = value / adc_max;
double Rt = bruch * R0 / (1 - bruch);
double log = Math.Log(Rt / Rn);
double T = 0;
if ((vaue > 1) && (value < 1023)) {
T = (BK * TnK) / (BK + Math.Log(Rt / Rn) * TnK)- 273;
}
return T;
}
```

2.11.11 Reading out the following error

Parameter

Symbol	Permissible values	Writable	Data type	Default value
`':Capt_lFollow'`	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 controllers with dsp drive

2.12.1 Setting the P component of the current controller at standstill

Parameter

Symbol	Permissible values	Writable	Data type	Default value
':dspdrive_KP_low'	0 to 1000	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 for controllers with dsp drive when at a standstill.

Normally, no change necessary.

Reading out

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

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

Parameter

Symbol	Permissible values	Writable	Data type	Default value
':dspdrive_KP_hig'	0 to 1000	Yes	u16 (integer)	10

Firmware response

Confirms the command through an echo.

Description

This parameter can be used to set the P component of the current controller for controllers with dsp drive during the run.

Normally, no change necessary.

Reading out

If the keyword is sent without a '= + Wert', 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

Parameter

Symbol	Permissible values	Writable	Data type	Default value
`':dspdrive_KP_scale'`	0 to 1000	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 for controllers with dsp drive during the run.

Normally, no change necessary.

Reading out

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

2.12.4 Setting the I component of the current controller at standstill

Parameter

Symbol	Permissible values	Writable	Data type	Default value
`':dspdrive_KI_low'`	0 to 1000	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 for controllers with dsp drive at standstill.

Normally, no change necessary.

Reading out

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

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

Parameter

Symbol	Permissible values	Writable	Data type	Default value
`':dspdrive_KI_hig'`	0 to 1000	Yes	u16 (integer)	10

Firmware response

Confirms the command through an echo.

Description

This parameter can be used to set the I component of the current controller for controllers with dsp drive during the run.

Normally, no change necessary.

Reading out

If the keyword is sent without a `'= + Wert'`, 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

Parameter

Symbol	Permissible values	Writable	Data type	Default value
`':dspdrive_KI_scale'`	0 to 1000	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 for controllers with dsp drive during the run.

Normally, no change necessary.

Reading out

If the keyword is sent without a `'= + Wert'`, 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 hardware that is used, the current VM is subject to the following restrictions:

- The available programming memory in the controller depends on the firmware version.
- 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

Prerequisites

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

- NanoJEasy programming environment installed
- SMCI47-S
- SMCP33
- SMCI33
- SMCI35
- SMCI36
- SMCI12
- PD6-N
- PD4-N

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.

Note:
In general, serial commands should only be used if the Java program is not actively acting on the controller at the time.

3.2 Command overview

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

capture commands

cl commands

comm commands

config commands

drive commands

dspdrive commands

io commands

util commands

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:

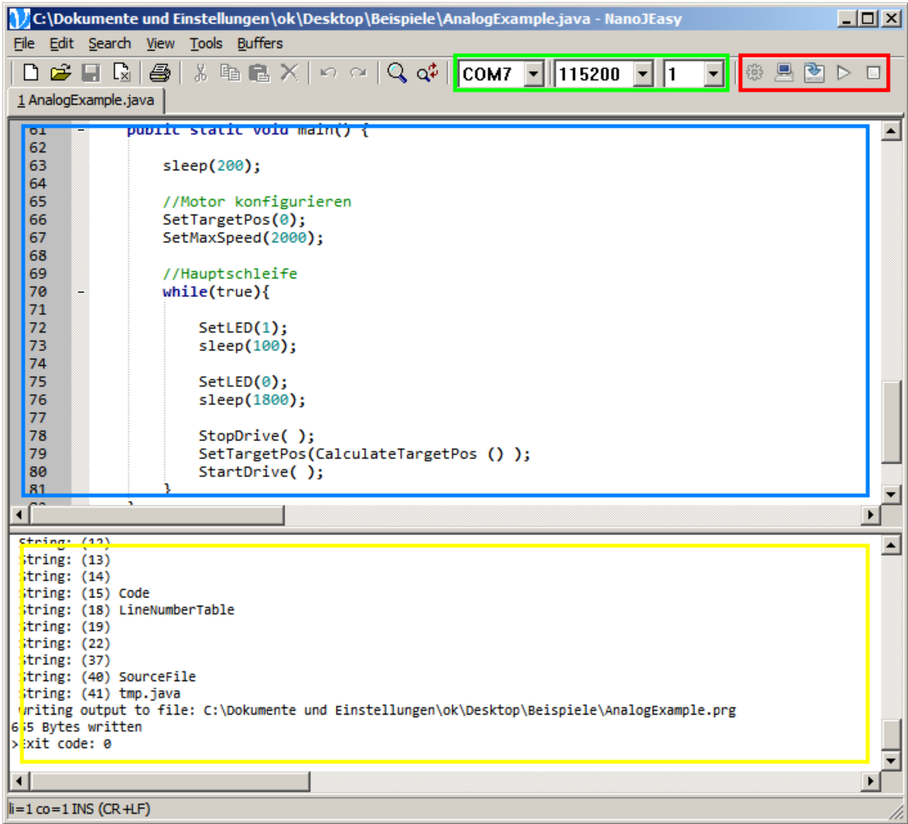
Step	Implementation
1	Double-click on the setup.exe file.
2	Select the desired language.
3	Confirm that you accept the license conditions.
4	Select the folder in which NanoJEasy should be installed.
5	Confirm or change the recommended start menu entry for NanoJEasy.
6	Start the installation.

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 six different classes which can be integrated in the source code.

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

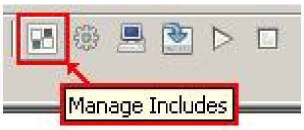
Integrating a class

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

```
import nanotec.*;
```

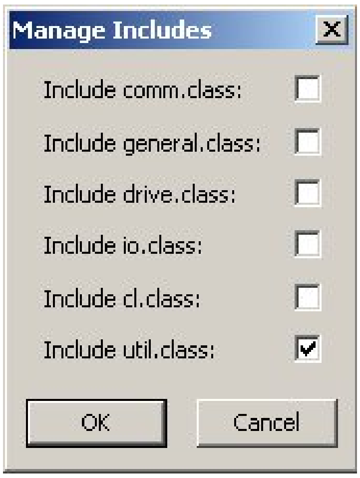
In addition, the classes which are really included on transfer to the controller must be selected in NanoJEasy.

“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 der Klasse].[Name der Funktion]();
```

Example:

```
drive.StartDrive();
```

3.5 Classes and functions

3.5.1 “capture” class

Application

The capture class is used to configure the scope mode. The following functions can be used to configure the controller in such a way that it determines control variables and sends these via the serial interface. See also Section 2.11.

capture.SetCaptTime

Declaration:

```
static native void SetCaptTime(int time);
```

This function sets the sample rate.

The function corresponds to the serial command `':Capt_Time<time>'`, see command *2.11.2 Setting the* sample rate.

Contained in firmware versions later than 15 March 2010.

capture.GetCaptTime

Declaration:

```
static native void GetCaptTime(int time);
```

This function reads the sample rate.

The function corresponds to the serial command `':Capt_Time'`, see command *2.11.2 Setting the* sample rate.

Contained in firmware versions later than 15 March 2010.

capture.SetCaptsPos

Declaration:

```
static native void SetCaptsPos(int value);
```

This function selects/deselects the setpoint position.

The function corresponds to the serial command `':Capt_sPos<value>'`, see command *2.11.3 Reading out the setpoint* position of the ramp generator.

Contained in firmware versions later than 15 March 2010.

capture.GetCaptsPos

Declaration:

```
static native int GetCaptsPos();
```

This function reads out whether the setpoint position is selected or not.

The function corresponds to the serial command `':Capt_sPos'`, see command *2.11.3 Reading out the setpoint* position of the ramp generator.

Contained in firmware versions later than 15 March 2010.

capture.SetCaptiPos

Declaration:

```
static native void SetCaptiPos(int value);
```

This function selects/deselects the actual position.

The function corresponds to the serial command `':Capt_iPos<value>'`, see command *2.11.4 Reading out the actual* position of the encoder.

Contained in firmware versions later than 15 March 2010.

capture.GetCaptiPos

Declaration:

```
static native int GetCaptiPos();
```

This function reads out whether the actual position is selected or not.

The function corresponds to the serial command `':Capt_iPos'`, see command *2.11.4 Reading out the actual* position of the encoder.

Contained in firmware versions later than 15 March 2010.

capture.SetCaptsCurr

Declaration:

```
static native void SetCaptsCurr(int value);
```

This function selects/deselects the set current.

The function corresponds to the serial command `':Capt_sCurr<value>'`, see command *2.11.5 Reading out the setpoint* current of the motor controller.

Contained in firmware versions later than 15 March 2010.

capture.GetCaptsCurr

Declaration:

```
static native int GetCaptsCurr();
```

This function reads out whether the set current is selected or not.

The function corresponds to the serial command `':Capt_sCurr'`, see command *2.11.5 Reading out the setpoint* current of the motor controller.

Contained in firmware versions later than 15 March 2010.

capture.SetCaptiVolt

Declaration:

```
static native void SetCaptiVolt(int value);
```

This function selects/deselects the current voltage.

The function corresponds to the serial command `':Capt_iVolt<value>'`, see command *2.11.6 Reading out the actual* voltage of the controller.

Contained in firmware versions later than 15 March 2010.

capture.GetCaptiVolt

Declaration:

```
static native int GetCaptiVolt();
```

This function reads out whether the actual current is selected or not.

The function corresponds to the serial command `':Capt_iVolt'`, see command *2.11.6 Reading out the actual* voltage of the controller.

Contained in firmware versions later than 15 March 2010.

capture.SetCaptiIn

Declaration:

```
static native void SetCaptiln(int value);
```

This function selects/deselects the bit mask of the inputs.

The function corresponds to the serial command `':Capt_iln<value>'`, see command *2.11.7 Reading out* the digital inputs.

Contained in firmware versions later than 15 March 2010.

capture.GetCaptiIn

Declaration:

```
static native int GetCaptiln();
```

This function reads out whether the bit mask of the inputs is selected or not.

The function corresponds to the serial command `':Capt_iln'`, see command *2.11.7 Reading out* the digital inputs.

Contained in firmware versions later than 15 March 2010.

capture.SetCaptiAnalog

Declaration:

```
static native void SetCaptiAnalog(int value);
```

This function selects/deselects the voltage at the analog input.

The function corresponds to the serial command `':Capt_iAnalog<value>'`, see command *2.11.8 Reading out the voltage* at the analog input.

Contained in firmware versions later than 15 March 2010.

capture.GetCaptiAnalog

Declaration:

```
static native int GetCaptiAnalog();
```

This function reads out whether the voltage at the analog input is selected or not.

The function corresponds to the serial command `':Capt_iAnalog'`, see command *2.11.8 Reading out the voltage* at the analog input.

Contained in firmware versions later than 15 March 2010.

capture.SetCaptiBus

Declaration:

```
static native void SetCaptiBus(int value);
```

This function selects/deselects the utilization of the CAN bus.

The function corresponds to the serial command `':Capt_iBus<value>'`, see command *2.11.9 Reading out* the CAN bus load.

Contained in firmware versions later than 15 March 2010.

capture.GetCaptiBus

Declaration:

```
static native int GetCaptiBus();
```

This function reads out whether the utilization of the CAN bus is selected or not.

The function corresponds to the serial command `':Capt_iBus'`, see command *2.11.9 Reading out* the CAN bus load.

Contained in firmware versions later than 15 March 2010.

capture.SetCaptlTemp

Declaration:

```
static native void SetCaptlTemp(int value);
```

This function selects/deselects the temperature of the controller.

The function corresponds to the serial command `':Capt_lTemp<value>'`, see command *2.11.10 Reading out the controller* temperature.

Contained in firmware versions later than 15 March 2010.

capture.GetCaptlTemp

Declaration:

```
static native int GetCaptlTemp();
```

This function reads out whether the temperature of the controller is selected or not.

The function corresponds to the serial command `':Capt_lTemp'`, see command *2.11.10 Reading out the controller* temperature.

Contained in firmware versions later than 15 March 2010.

capture.SetCaptlFollow

Declaration:

```
static native void SetCaptlFollow(int value);
```

This function selects/deselects the following error.

The function corresponds to the serial command `':Capt_lFollow<offset>'`, see command *2.11.11 Reading out* the following error.

Contained in firmware versions later than 15 March 2010.

capture.GetCaptlFollow

Declaration:

```
static native int GetCaptlFollow();
```

This function reads out whether the following error of the controller is selected or not.

The function corresponds to the serial command `':Capt_IFollow'`, see command *2.11.11 Reading out* the following error.

Contained in firmware versions later than 15 March 2010.

3.5.2 "cl" class

Application

The cl class is used to configure the closed loop. The PID parameters can be set and the closed loop status can be manipulated.

cl.SetClosedLoop

Declaration:

```
static native void SetClosedLoop(int value);
```

This function activates/deactivates the control loop. The mode is not activated until an internal reference run has been performed or until more than one rotation has been travelled with auto enable activated.

The function corresponds to the serial command `':CL_enable<value>'`, see command *2.9.1 Activating closed-loop* mode.

Contained in firmware versions later than 15 March 2010.

cl.GetClosedLoop

Declaration:

```
static native int GetClosedLoop();
```

This function reads out whether the control loop is activated/deactivated.

The function corresponds to the serial command `':CL_enable'`, see command *2.9.1 Activating closed-loop* mode.

Contained in firmware versions later than 15 March 2010.

cl.IsClosedLoopEnabled

Declaration:

```
static native int IsClosedLoopEnabled();
```

This function reads out whether the control loop is activated/deactivated.

- Value 0: control loop is not active
- Value 1: control loop is active (only if the special reference run was performed)

The function corresponds to the serial command `':CL_is_enabled'`, see command *2.9.2 Reading out the* closed loop mode status.

Contained in firmware versions later than 15 March 2010.

cl.SetKPvZ

Declaration:

```
static native void SetKPvZ(int value);
```

This function sets the numerator of the P component of the speed controller.

The function corresponds to the serial command `':CL_KP_v_Z<value>'`, see command *2.9.13 Setting the numerator of the P* component of the speed controller.

Contained in firmware versions later than 15 March 2010.

cl.GetKPvZ

Declaration:

```
static native int GetKPvZ();
```

This function reads out the numerator of the P component of the speed controller.

The function corresponds to the serial command `':CL_KP_v_Z'`, see command *2.9.13 Setting the numerator of the P* component of the speed controller.

Contained in firmware versions later than 15 March 2010.

cl.SetKPvN

Declaration:

```
static native void SetKPvN(int value);
```

This function sets the denominator of the P component of the speed controller.

The function corresponds to the serial command `':CL_KP_v_N<value>'`, see command *2.9.14 Setting the denominator of the P* component of the speed controller.

Contained in firmware versions later than 15 March 2010.

cl.GetKPvN

Declaration:

```
static native int GetKPvN();
```

This function reads out the denominator of the P component of the speed controller.

The function corresponds to the serial command `':CL_KP_v_N'`, see command *2.9.14 Setting the denominator of the P* component of the speed controller.

Contained in firmware versions later than 15 March 2010.

cl.SetKIvZ

Declaration:

```
static native void SetKIvZ(int value);
```

This function sets the numerator of the I component of the speed controller.

The function corresponds to the serial command `':CL_KI_v_Z<value>'`, see command *2.9.15 Setting the numerator of the I* component of the speed controller.

Contained in firmware versions later than 15 March 2010.

cl.GetKIvZ

Declaration:

```
static native int GetKIvZ();
```

This function reads out the numerator of the I component of the speed controller.

The function corresponds to the serial command `':CL_KI_v_Z'`, see command *2.9.15 Setting the numerator of the I* component of the speed controller.

Contained in firmware versions later than 15 March 2010.

cl.SetKIvN

Declaration:

```
static native void SetKIvN(int value);
```

This function sets the denominator of the I component of the speed controller.

The function corresponds to the serial command `':CL_KI_v_N<value>'`, see command *2.9.16 Setting the denominator of the I* component of the speed controller.

Contained in firmware versions later than 15 March 2010.

cl.GetKIvN

Declaration:

```
static native int GetKIvN();
```

This function reads out the denominator of the I component of the speed controller.

The function corresponds to the serial command `':CL_KI_v_N'`, see command *2.9.16 Setting the denominator of the I* component of the speed controller.

Contained in firmware versions later than 15 March 2010.

cl.SetKDvZ

Declaration:

```
static native void SetKDvZ(int value);
```

This function sets the numerator of the D component of the speed controller.

The function corresponds to the serial command `':CL_KD_v_Z<value>'`, see command *2.9.17 Setting the numerator of the D* component of the speed controller.

Contained in firmware versions later than 15 March 2010.

cl.GetKDvZ

Declaration:

```
static native int GetKDvZ();
```

This function reads out the numerator of the D component of the speed controller.

The function corresponds to the serial command `':CL_KD_v_Z'`, see command *2.9.17 Setting the numerator of the D* component of the speed controller.

Contained in firmware versions later than 15 March 2010.

cl.SetKDvN

Declaration:

```
static native void SetKDvN(int value);
```

This function sets the denominator of the D component of the speed controller.

The function corresponds to the serial command `':CL_KD_v_N<value>'`, see command *2.9.18 Setting the denominator of the D* component of the speed controller.

Contained in firmware versions later than 15 March 2010.

cl.GetKDvN

Declaration:

```
static native int GetKDvN();
```

This function reads out the denominator of the D component of the speed controller.

The function corresponds to the serial command `':CL_KD_v_N'`, see command *2.9.18 Setting the denominator of the D* component of the speed controller.

Contained in firmware versions later than 15 March 2010.

cl.SetKPsZ

Declaration:

```
static native void SetKPsZ(int value);
```

This function sets the numerator of the P component of the position controller.

The function corresponds to the serial command `':CL_KP_s_Z<value>'`, see command *2.9.25 Setting the numerator of the P* component of the position controller.

Contained in firmware versions later than 15 March 2010.

cl.GetKPsZ

Declaration:

```
static native int GetKPsZ();
```

This function reads out the numerator of the P component of the position controller.

The function corresponds to the serial command `':CL_KP_s_Z'`, see command *2.9.25 Setting the numerator of the P* component of the position controller.

Contained in firmware versions later than 15 March 2010.

cl.SetKPsN

Declaration:

```
static native void SetKPsN(int value);
```

This function sets the denominator of the P component of the position controller.

The function corresponds to the serial command `':CL_KP_s_N<value>'`, see command *2.9.26 Setting the denominator of the P* component of the position controller.

Contained in firmware versions later than 15 March 2010.

cl.GetKPsN

Declaration:

```
static native int GetKPsN();
```

This function reads out the denominator of the P component of the position controller.

The function corresponds to the serial command `':CL_KP_s_N'`, see command *2.9.26 Setting the denominator of the P* component of the position controller.

Contained in firmware versions later than 15 March 2010.

cl.SetKIsZ

Declaration:

```
static native void SetKIsZ(int value);
```

This function sets the numerator of the I component of the position controller.

The function corresponds to the serial command `':CL_KI_s_Z<value>'`, see command *2.9.27 Setting the numerator of the I* component of the position controller.

Contained in firmware versions later than 15 March 2010.

cl.GetKIsZ

Declaration:

```
static native int GetKIsZ();
```

This function reads out the numerator of the I component of the position controller.

The function corresponds to the serial command `':CL_KI_s_Z'`, see command *2.9.27 Setting the numerator of the I* component of the position controller.

Contained in firmware versions later than 15 March 2010.

cl.SetKIsN

Declaration:

```
static native void SetKIsN(int value);
```

This function sets the denominator of the I component of the position controller.

The function corresponds to the serial command `':CL_KI_s_N<value>'`, see command *2.9.28 Setting the denominator of the I* component of the position controller.

Contained in firmware versions later than 15 March 2010.

cl.GetKIsN

Declaration:

```
static native int GetKIsN();
```

This function reads out the denominator of the I component of the position controller.

The function corresponds to the serial command `':CL_KI_s_N'`, see command *2.9.28 Setting the denominator of the I* component of the position controller.

Contained in firmware versions later than 15 March 2010.

cl.SetKDsZ

Declaration:

```
static native void SetKDsZ(int value);
```

This function sets the numerator of the D component of the position controller.

The function corresponds to the serial command `':CL_KD_s_Z<value>'`, see command *2.9.29 Setting the numerator of the D* component of the position controller.

Contained in firmware versions later than 15 March 2010.

cl.GetKDsZ

Declaration:

```
static native int GetKDsZ();
```

This function reads out the numerator of the D component of the position controller.

The function corresponds to the serial command `':CL_KD_s_Z'`, see command *2.9.29 Setting the numerator of the D* component of the position controller.

Contained in firmware versions later than 15 March 2010.

cl.SetKDsN

Declaration:

```
static native void SetKDsN(int value);
```

This function sets the denominator of the D component of the position controller.

The function corresponds to the serial command `':CL_KD_s_N<value>'`, see command *2.9.30 Setting the denominator of the D* component of the position controller.

Contained in firmware versions later than 15 March 2010.

cl.GetKDsN

Declaration:

```
static native int GetKDsN();
```

This function reads out the denominator of the D component of the position controller.

The function corresponds to the serial command `':CL_KD_s_N'`, see command *2.9.30 Setting the denominator of the D* component of the position controller.

Contained in firmware versions later than 15 March 2010.

cl.SetKPcsvZ

Declaration:

```
static native void SetKPcsvZ(int value);
```

This function sets the numerator of the P component of the cascading speed controller.

The function corresponds to the serial command `':CL_KP_csv_Z<value>'`, see command *2.9.19 Setting the numerator of the P component* of the cascading speed controller.

Contained in firmware versions later than 15 March 2010.

cl.GetKPcsvZ

Declaration:

```
static native int GetKPcsvZ();
```

This function reads out the numerator of the P component of the cascading speed controller.

The function corresponds to the serial command `':CL_KP_csv_Z'`, see command *2.9.19 Setting the numerator of the P component* of the cascading speed controller.

Contained in firmware versions later than 15 March 2010.

cl.SetKPcsvN

Declaration:

```
static native void SetKPcsvN(int value);
```

This function sets the denominator of the P component of the cascading speed controller.

The function corresponds to the serial command `':CL_KP_csv_N<value>'`, see command *2.9.20 Setting the denominator of the P component* of the cascading speed controller.

Contained in firmware versions later than 15 March 2010.

cl.GetKPcsvN

Declaration:

```
static native int GetKPcsvN();
```

This function reads out the denominator of the P component of the cascading speed controller.

The function corresponds to the serial command `':CL_KP_csv_N'`, see command *2.9.20 Setting the denominator of the P component* of the cascading speed controller.

Contained in firmware versions later than 15 March 2010.

cl.SetKIcsvZ

Declaration:

```
static native void SetKIcsvZ(int value);
```

This function sets the numerator of the I component of the cascading speed controller.

The function corresponds to the serial command `':CL_KI_csv_Z<value>'`, see command *2.9.21 Setting the numerator of the I component* of the cascading speed controller.

Contained in firmware versions later than 15 March 2010.

cl.GetKIcsvZ

Declaration:

```
static native int GetKIcsvZ();
```

This function reads out the numerator of the I component of the cascading speed controller.

The function corresponds to the serial command `':CL_KI_csv_Z'`, see command *2.9.21 Setting the numerator of the I component* of the cascading speed controller.

Contained in firmware versions later than 15 March 2010.

cl.SetKIcsvN

Declaration:

```
static native void SetKIcsvN(int value);
```

This function sets the denominator of the I component of the cascading speed controller.

The function corresponds to the serial command `':CL_KI_csv_N<value>'`, see command *2.9.22 Setting the denominator of the I component* of the cascading speed controller.

Contained in firmware versions later than 15 March 2010.

cl.GetKIcsvN

Declaration:

```
static native int GetKIcsvN();
```

This function reads out the denominator of the I component of the cascading speed controller.

The function corresponds to the serial command `':CL_KI_csv_N'`, see command *2.9.22 Setting the denominator of the I component* of the cascading speed controller.

Contained in firmware versions later than 15 March 2010.

cl.SetKDcsvZ

Declaration:

```
static native void SetKDcsvZ(int value);
```

This function sets the numerator of the D component of the cascading speed controller.

The function corresponds to the serial command `':CL_KD_csv_Z<value>'`, see command *2.9.23 Setting the numerator of the D component* of the cascading speed controller.

Contained in firmware versions later than 15 March 2010.

cl.GetKDcsvZ

Declaration:

```
static native int GetKDcsvZ();
```

This function reads out the numerator of the D component of the cascading speed controller.

The function corresponds to the serial command `':CL_KD_csv_Z'`, see command *2.9.23 Setting the numerator of the D component* of the cascading speed controller.

Contained in firmware versions later than 15 March 2010.

cl.SetKDcsvN

Declaration:

```
static native void SetKDcsvN(int value);
```

This function sets the denominator of the D component of the cascading speed controller.

The function corresponds to the serial command `':CL_KD_csv_N<value>'`, see command *2.9.24 Setting the denominator of the D component* of the cascading speed controller.

Contained in firmware versions later than 15 March 2010.

cl.GetKDcsvN

Declaration:

```
static native int GetKDcsvN();
```

This function reads out the denominator of the D component of the cascading speed controller.

The function corresponds to the serial command `':CL_KD_csv_N'`, see command *2.9.24 Setting the denominator of the D component* of the cascading speed controller.

Contained in firmware versions later than 15 March 2010.

cl.SetKPcssZ

Declaration:

```
static native void SetKPcssZ(int value);
```

This function sets the numerator of the P component of the cascading position controller.

The function corresponds to the serial command `':CL_KP_css_Z<value>'`, see command *2.9.31 Setting the numerator of the P component* of the cascading position controller.

Contained in firmware versions later than 15 March 2010.

cl.GetKPcssZ

Declaration:

```
static native int GetKPcssZ();
```

This function reads out the numerator of the P component of the cascading position controller.

The function corresponds to the serial command `':CL_KP_css_Z'`, see command *2.9.31 Setting the numerator of the P component* of the cascading position controller.

Contained in firmware versions later than 15 March 2010.

cl.SetKPcssN

Declaration:

```
static native void SetKPcssN(int value);
```

This function sets the denominator of the P component of the cascading position controller.

The function corresponds to the serial command `':CL_KP_css_N<value>'`, see command *2.9.32 Setting the denominator of the P component* of the cascading position controller.

Contained in firmware versions later than 15 March 2010.

cl.GetKPcssN

Declaration:

```
static native int GetKPcssN();
```

This function reads out the denominator of the P component of the cascading position controller.

The function corresponds to the serial command `':CL_KP_css_N'`, see command *2.9.32 Setting the denominator of the P component* of the cascading position controller.

Contained in firmware versions later than 15 March 2010.

cl.SetKIcssZ

Declaration:

```
static native void SetKIcssZ(int value);
```

This function sets the numerator of the I component of the cascading position controller.

The function corresponds to the serial command `':CL_KI_css_Z<value>'`, see command *2.9.33 Setting the numerator of the I component* of the cascading position controller.

Contained in firmware versions later than 15 March 2010.

cl.GetKIcssZ

Declaration:

```
static native int GetKIcssZ();
```

This function reads out the numerator of the I component of the cascading position controller.

The function corresponds to the serial command `':CL_KI_css_Z'`, see command *2.9.33 Setting the numerator of the I component* of the cascading position controller.

Contained in firmware versions later than 15 March 2010.

cl.SetKIcssN

Declaration:

```
static native void SetKIcssN(int value);
```

This function sets the denominator of the I component of the cascading position controller.

The function corresponds to the serial command `':CL_KI_css_N<value>'`, see command *2.9.34 Setting the denominator of the I component* of the cascading position controller.

Contained in firmware versions later than 15 March 2010.

cl.GetKIcssN

Declaration:

```
static native int GetKIcssN();
```

This function reads out the denominator of the I component of the cascading position controller.

The function corresponds to the serial command `':CL_KI_css_N'`, see command *2.9.34 Setting the denominator of the I component* of the cascading position controller.

Contained in firmware versions later than 15 March 2010.

cl.SetKDcssZ

Declaration:

```
static native void SetKDcssZ(int value);
```

This function sets the numerator of the D component of the cascading position controller.

The function corresponds to the serial command `':CL_KD_css_Z<value>'`, see command *2.9.35 Setting the numerator of the D component* of the cascading position controller.

Contained in firmware versions later than 15 March 2010.

cl.GetKDcssZ

Declaration:

```
static native int GetKDcssZ();
```

This function reads out the numerator of the D component of the cascading position controller.

The function corresponds to the serial command `':CL_KD_css_Z'`, see command *2.9.35 Setting the numerator of the D component* of the cascading position controller.

Contained in firmware versions later than 15 March 2010.

cl.SetKDcssN

Declaration:

```
static native void SetKDcssN(int value);
```

This function sets the denominator of the D component of the cascading position controller.

The function corresponds to the serial command `':CL_KD_css_N<value>'`, see command *2.9.36 Setting the denominator of the D component* of the cascading position controller.

Contained in firmware versions later than 15 March 2010.

cl.GetKDcssN

Declaration:

```
static native int GetKDcssN();
```

This function reads out the denominator of the D component of the cascading position controller.

The function corresponds to the serial command `':CL_KD_css_N'`, see command *2.9.36 Setting the denominator of the D component* of the cascading position controller.

Contained in firmware versions later than 15 March 2010.

cl.SetPositionWindow

Declaration:

```
static native void SetPositionWindow(int value);
```

This function sets the tolerance window for the end position in the closed loop mode.

The function corresponds to the serial command `':CL_position_window<value>'`, see command *2.9.4 Setting the tolerance window* for the limit position.

Contained in firmware versions later than 15 March 2010.

cl.GetPositionWindow

Declaration:

```
static native int GetPositionWindow();
```

This function reads out the tolerance window for the end position in the closed loop mode.

The function corresponds to the serial command `':CL_position_window'`, see command *2.9.4 Setting the tolerance window* for the limit position.

Contained in firmware versions later than 15 March 2010.

cl.SetPositionWindowTime

Declaration:

```
static native void SetPositionWindowTime(int time);
```

This function sets the time for the tolerance window of the end position in the closed loop mode.

The function corresponds to the serial command `':CL_position_window_time<time>'`, see command *2.9.5 Setting the time for the tolerance* window of the limit position.

Contained in firmware versions later than 15 March 2010.

cl.GetPositionWindowTime

Declaration:

```
static native int GetPositionWindowTime();
```

This function reads out the time for the tolerance window of the end position in the closed loop mode.

The function corresponds to the serial command `':CL_position_window_time'`, see command *2.9.5 Setting the time for the tolerance* window of the limit position.

Contained in firmware versions later than 15 March 2010.

cl.SetFollowingErrorWindow

Declaration:

```
static native void SetFollowingErrorWindow(int value);
```

This function sets the maximum allowed following error in the closed loop mode.

The function corresponds to the serial command `':CL_following_error_window<value>'`, see command *2.9.6 Setting the maximum allowed* following error.

Contained in firmware versions later than 15 March 2010.

cl.GetFollowingErrorWindow

Declaration:

```
static native int GetFollowingErrorWindow();
```

This function reads out the maximum allowed following error in the closed loop mode.

The function corresponds to the serial command `':CL_following_error_window'`, see command *2.9.6 Setting the maximum allowed* following error.

Contained in firmware versions later than 15 March 2010.

cl.SetFollowingErrorTimeout

Declaration:

```
static native void SetFollowingErrorTimeout(int time);
```

This function sets the time for the maximum allowed following error in the closed loop mode.

The function corresponds to the serial command `':CL_following_error_timeout<time>'`, see command *2.9.7 Setting the time for the* maximum following error.

Contained in firmware versions later than 15 March 2010.

cl.GetFollowingErrorTimeout

Declaration:

```
static native int GetFollowingErrorTimeout();
```

This function reads out the time for the maximum allowed following error in the closed loop mode.

The function corresponds to the serial command `':CL_following_error_timeout'`, see command *2.9.7 Setting the time for the* maximum following error.

Contained in firmware versions later than 15 March 2010.

cl.SetSpeedErrorWindow

Declaration:

```
static native void SetSpeedErrorWindow(int value);
```

This function sets the maximum allowed speed deviation in the closed loop mode.

The function corresponds to the serial command `':CL_speed_error_window<value>'`, see command *2.9.8 Maximum speed deviation*.

Contained in firmware versions later than 15 March 2010.

cl.GetSpeedErrorWindow

Declaration:

```
static native int GetSpeedErrorWindow();
```

This function reads out the maximum allowed speed deviation in the closed loop mode.

The function corresponds to the serial command `':CL_speed_error_window'`, see command *2.9.8 Maximum speed deviation*.

Contained in firmware versions later than 15 March 2010.

cl.SetSpeedErrorTimeout

Declaration:

```
static native void SetSpeedErrorTimeout(int time);
```

This function sets the time for the maximum allowed speed deviation in the closed loop mode.

The function corresponds to the serial command `':CL_speed_error_timeout<time>'`, see command *2.9.9 Time for maximum speed deviation*.

Contained in firmware versions later than 15 March 2010.

cl.GetSpeedErrorTimeout

Declaration:

```
static native int GetSpeedErrorTimeout();
```

This function reads out the time for the maximum allowed speed deviation in the closed loop mode.

The function corresponds to the serial command `':CL_speed_error_timeout'`, see command *2.9.9 Time for maximum speed deviation*.

Contained in firmware versions later than 15 March 2010.

cl.SetCLLoadAngle1

Declaration:

```
static native void SetCLLoadAngle1(int value);
```

This function sets the load angle 1 of the motor from the closed loop test run.

The function corresponds to the serial command ':CL_la_a<value>', see command *2.10.2 Setting/reading out load angle* measurement values of the motor.

Contained in firmware versions later than 15 March 2010.

cl.GetCLLoadAngle1

Declaration:

```
static native int GetCLLoadAngle1();
```

This function reads out the load angle 1 of the motor from the closed loop test run.

The function corresponds to the serial command ':CL_la_a', see command *2.10.2 Setting/reading out load angle* measurement values of the motor.

Contained in firmware versions later than 15 March 2010.

cl.SetCLLoadAngle2

Declaration:

```
static native void SetCLLoadAngle2(int value);
```

This function sets the load angle 2 of the motor from the closed loop test run.

The function corresponds to the serial command ':CL_la_b<value>', see command *2.10.2 Setting/reading out load angle* measurement values of the motor.

Contained in firmware versions later than 15 March 2010.

cl.GetCLLoadAngle2

Declaration:

```
static native int GetCLLoadAngle2();
```

This function reads out the load angle 1 of the motor from the closed loop test run.

The function corresponds to the serial command ':CL_la_b', see command *2.10.2 Setting/reading out load angle* measurement values of the motor.

Contained in firmware versions later than 15 March 2010.

cl.SetCLLoadAngle3

Declaration:

```
static native void SetCLLoadAngle3(int value);
```

This function sets the load angle 3 of the motor from the closed loop test run.

The function corresponds to the serial command ':CL_la_c<value>', see command *2.10.2 Setting/reading out load angle* measurement values of the motor.

Contained in firmware versions later than 15 March 2010.

cl.GetCLLoadAngle3

Declaration:

```
static native int GetCLLoadAngle3();
```

This function read out the load angle 3 of the motor from the closed loop test run.

The function corresponds to the serial command `':CL_la_c'`, see command *2.10.2 Setting/reading out load angle* measurement values of the motor.

Contained in firmware versions later than 15 March 2010.

cl.SetCLLoadAngle4

Declaration:

```
static native void SetCLLoadAngle4(int value);
```

This function sets the load angle 4 of the motor from the closed loop test run.

The function corresponds to the serial command `':CL_la_d<value>'`, see command *2.10.2 Setting/reading out load angle* measurement values of the motor.

Contained in firmware versions later than 15 March 2010.

cl.GetCLLoadAngle4

Declaration:

```
static native int GetCLLoadAngle4();
```

This function reads out the load angle 4 of the motor from the closed loop test run.

The function corresponds to the serial command `':CL_la_d'`, see command *2.10.2 Setting/reading out load angle* measurement values of the motor.

Contained in firmware versions later than 15 March 2010.

cl.SetCLLoadAngle5

Declaration:

```
static native void SetCLLoadAngle5(int value);
```

This function sets the load angle 5 of the motor from the closed loop test run.

The function corresponds to the serial command `':CL_la_e<value>'`, see command *2.10.2 Setting/reading out load angle* measurement values of the motor.

Contained in firmware versions later than 15 March 2010.

cl.GetCLLoadAngle5

Declaration:

```
static native int GetCLLoadAngle5();
```

This function reads out the load angle 5 of the motor from the closed loop test run.

The function corresponds to the serial command `':CL_la_e'`, see command *2.10.2 Setting/reading out load angle* measurement values of the motor.

Contained in firmware versions later than 15 March 2010.

cl.SetCLLoadAngle6

Declaration:

```
static native void SetCLLoadAngle6(int value);
```

This function sets the load angle 6 of the motor from the closed loop test run.

The function corresponds to the serial command `':CL_la_f<value>'`, see command *2.10.2 Setting/reading out load angle* measurement values of the motor.

Contained in firmware versions later than 15 March 2010.

cl.GetCLLoadAngle6

Declaration:

```
static native int GetCLLoadAngle6();
```

This function reads out the load angle 6 of the motor from the closed loop test run.

The function corresponds to the serial command `':CL_la_f'`, see command *2.10.2 Setting/reading out load angle* measurement values of the motor.

Contained in firmware versions later than 15 March 2010.

cl.SetCLLoadAngle7

Declaration:

```
static native void SetCLLoadAngle7(int value);
```

This function sets the load angle 7 of the motor from the closed loop test run.

The function corresponds to the serial command `':CL_la_g<value>'`, see command *2.10.2 Setting/reading out load angle* measurement values of the motor.

Contained in firmware versions later than 15 March 2010.

cl.GetCLLoadAngle7

Declaration:

```
static native int GetCLLoadAngle7();
```

This function reads out the load angle 7 of the motor from the closed loop test run.

The function corresponds to the serial command `':CL_la_g'`, see command *2.10.2 Setting/reading out load angle* measurement values of the motor.

Contained in firmware versions later than 15 March 2010.

cl.SetCLNodeDistance

Declaration:

```
static native void SetCLNodeDistance(int value);
```

This function sets the sampling point spacing for the load angle curve.

The function corresponds to the serial command `':CL_la_node_distance<value>'`, see command *2.9.37 Setting the sampling point* spacing of the load angle curve.

Contained in firmware versions later than 15 March 2010.

cl.GetCLNodeDistance

Declaration:

```
static native int GetCLNodeDistance();
```

This function reads out the sampling point spacing for the load angle curve.

The function corresponds to the serial command `':CL_la_node_distance'`, see command *2.9.37 Setting the sampling point* spacing of the load angle curve.

Contained in firmware versions later than 15 March 2010.

cl.SetCLPoscntOffset

Declaration:

```
static native void SetCLPoscntOffset(int offset);
```

This function sets the of fset between the encoder and the motor.

The function corresponds to the serial command `':CL_poscnt_offset<offset>'`, see command *2.10.1 Reading out the* encoder/motor offset.

Contained in firmware versions later than 15 March 2010.

cl.GetCLPoscntOffset

Declaration:

```
static native int GetCLPoscntOffset();
```

This function reads out the offset between the encoder and the motor determined during the test run.

The function corresponds to the serial command `':CL_poscnt_offset'`, see command *2.10.1 Reading out the* encoder/motor offset.

Contained in firmware versions later than 15 March 2010.

cl.GetVelocityActualValue

Declaration:

```
static native int GetVelocityActualValue();
```

This function reads out the current speed (only in closed loop mode).

The function corresponds to the serial command `':v'`, see command *2.7.11 Reading out* the speed.

Contained in firmware versions later than 15 March 2010.

3.5.3 “comm” class

Application

The comm class is used to configure serial communication and send data.

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.

comm.SetBaudrate

Declaration:

```
static native void SetBaudrate(int value);
```

This function sets the baud rate of the controller.

The function corresponds to the serial command `':baud<value>'`, see command *2.5.41 Setting baudrate of the* controller.

Contained in firmware versions later than 15 March 2010.

comm.GetBaudrate

Declaration:

```
static native int GetBaudrate();
```

This function reads out the baud rate of the controller.

The function corresponds to the serial command `':baud'`, see command *2.5.41 Setting baudrate of the* controller.

Contained in firmware versions later than 15 March 2010.

comm.SetCRC

Declaration:

```
static native void SetCRC(int value);
```

Switches the check of the serial communication using a CRC checksum (cyclic redundancy check) on or off:

- Value 0: CRC check deactivated
- Value 1: CRC check activated

The function corresponds to the serial command `':crc<value>'`, see command *2.5.42 Setting the* CRC checksum.

Contained in firmware versions later than 15 March 2010.

comm.GetCRC

Declaration:

```
static native int GetCRC();
```

This function reads out whether the check of the serial communication using a CRC checksum is switched on or off.

The function corresponds to the serial command `':crc'`, see command *2.5.42 Setting the* CRC checksum.

Contained in firmware versions later than 15 March 2010.

comm.SetSupressResponse

Declaration:

```
static native void SetSupressResponse(int value);
```

This function activates or deactivates the response suppression on sending.

- value = 0: response suppression on
- value = 1: response suppression off

The function corresponds to the serial command `'|<value>'`, see command *2.6.4 Reading out the* current record.

Contained in firmware versions later than 15 March 2010.

3.5.4 "config" class

Application

The config class is used to configure the general controller settings.

config.SetSendStatusWhenCompleted

Declaration:

```
static native void SetSendStatusWhenCompleted(int flag);
```

This function switches the independent sending of a status at the end of a run on/off.

- sendStatus = 0: automatic sending off
- sendStatus = 1: automatic sending on

The function corresponds to the serial command `'J<flag>'`, see command *2.5.32 Setting automatic sending of the* status.

Contained in firmware versions later than 15 March 2010.

config.GetSendStatusWhenCompleted

Declaration:

```
static native int GetSendStatusWhenCompleted();
```

This function reads whether the independent sending of a status at the end of a run is switched on.

- sendStatus = 0: automatic sending off
- sendStatus = 1: automatic sending on

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

Contained in firmware versions later than 15 March 2010.

config.SetRecordForAutoCorrect

Declaration:

```
static native void SetRecordForAutoCorrect(int record);
```

This function configures on the automatic error correction of the motor.

The function corresponds to the serial command `'F<record>'`, see command *2.5.11 Setting the record for* auto correction.

Contained in firmware versions later than 15 March 2010.

config.GetRecordForAutoCorrect

Declaration:

```
static native int GetRecordForAutoCorrect();
```

This function reads out which record is set for the automatic error correction.

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

Contained in firmware versions later than 15 March 2010.

config.SetEncoderDirection

Declaration:

```
static native void SetEncoderDirection(int value);
```

This function sets the encoder rotation direction. If the parameter value is 1, the direction of the rotary encoder is reversed.

The function corresponds to the serial command `'q<value>'`, see command *2.5.12 Setting the* encoder direction.

Contained in firmware versions later than 15 March 2010.

config.GetEncoderDirection

Declaration:

```
static native int GetEncoderDirection();
```

This function reads out whether the encoder rotation direction of the motor will be reversed.

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

Contained in firmware versions later than 15 March 2010.

config.SetSwingOutTime

Declaration:

```
static native void SetSwingOutTime(int time);
```

This function sets the swing out time.

The function corresponds to the serial command `'O<time>'`, see command *2.5.13 Setting the* swing out time.

Contained in firmware versions later than 15 March 2010.

config.GetSwingOutTime

Declaration:

```
static native int GetSwingOutTime();
```

This function reads out the swing out time.

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

Contained in firmware versions later than 15 March 2010.

config.SetAngleDeviationMax

Declaration:

```
static native void SetAngleDeviationMax(int value);
```

This function sets the maximum angle deviation between the setpoint position and the encoder value.

The function corresponds to the serial command `'X<value>'`, see command *2.5.14 Setting the maximum encoder* deviation.

Contained in firmware versions later than 15 March 2010.

config.GetAngleDeviationMax

Declaration:

```
static native int GetAngleDeviationMax();
```

This function reads out the maximum angle deviation between the setpoint position and the encoder value.

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

Contained in firmware versions later than 15 March 2010.

config.SetCurrentReductionTime

Declaration:

```
static native void SetCurrentReductionTime(int value);
```

This function sets the wait time at a standstill until the current is lowered.

The function corresponds to the serial command `'G<value>'`, see command *2.7.8 Adjusting the time until the* current reduction.

Contained in firmware versions later than 15 March 2010.

config.GetCurrentReductionTime

Declaration:

```
static native int GetCurrentReductionTime();
```

This function reads out the wait time at a standstill until the current is lowered.

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

Contained in firmware versions later than 15 March 2010.

config.SetReverseClearance

Declaration:

```
static native void SetReverseClearance(int value);
```

This function sets the reverse clearance in steps.

The function corresponds to the serial command `'z<value>'`, see command *2.5.34 Setting the* reverse clearance.

Contained in firmware versions later than 15 March 2010.

config.GetReverseClearance

Declaration:

```
static native int GetReverseClearance();
```

This function reads the reverse clearance in steps.

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

Contained in firmware versions later than 15 March 2010.

config.ResetEEProm

Declaration:

```
static native void ResetEEProm();
```

This function sets all settings of the controller back to default values (factory default settings).

The function corresponds to the serial command `'~'`, see command *2.5.31 Carrying out an* EEPROM reset.

Contained in firmware versions later than 15 March 2010.

ATTENTION: This function also deletes the Java program! The program continues running to the end (since in memory) but cannot be started again after that.

config.SetMotorPP

Declaration:

```
static native void SetMotorPP(int value);
```

This function sets the motor pole pair.

The function corresponds to the serial command `':CL_motor_pp<value>'`, see command *2.9.10 Setting the motor pole* pairs.

Contained in firmware versions later than 15 March 2010.

config.GetMotorPP

Declaration:

```
static native int GetMotorPP();
```

This function reads out the motor pole pair.

The function corresponds to the serial command `':CL_motor_pp'`, see command *2.9.10 Setting the motor pole* pairs.

Contained in firmware versions later than 15 March 2010.

config.SetRotencInc

Declaration:

```
static native void SetRotencInc(int value);
```

This function sets the number of encoder increments.

The function corresponds to the serial command `':CL_rotenc_inc<value>'`, see command *2.9.11 Setting the number of* increments.

Contained in firmware versions later than 15 March 2010.

config.GetRotencInc

Declaration:

```
static native int GetRotencInc();
```

This function reads out the number of encoder increments.

The function corresponds to the serial command `':CL_rotenc_inc'`, see command *2.9.11 Setting the number of* increments.

Contained in firmware versions later than 15 March 2010.

config.SetBrakeTA

Declaration:

```
static native void SetBrakeTA(int time);
```

This function sets the wait time for switching off the brake voltage.

The function corresponds to the serial command `':brake_ta<time>'`, see command *2.5.38 Setting the wait time for switching* off the brake voltage.

Contained in firmware versions later than 15 March 2010.

config.GetBrakeTA

Declaration:

```
static native int GetBrakeTA();
```

This function reads out the wait time for switching off the brake voltage.

The function corresponds to the serial command `':brake_ta'`, see command *2.5.38 Setting the wait time for switching* off the brake voltage.

Contained in firmware versions later than 15 March 2010.

config.SetBrakeTB

Declaration:

```
static native void SetBrakeTB(int time);
```

This function sets the time in milliseconds between switching off of the brake voltage and enabling of a motor movement.

The function corresponds to the serial command `':brake_tb<time>'`, see command *2.5.39 Setting the wait time* for the motor movement.

Contained in firmware versions later than 15 March 2010.

config.GetBrakeTB

Declaration:

```
static native int GetBrakeTB();
```

This function sets the time between switching off of the brake voltage and enabling of a motor movement.

The function corresponds to the serial command `':brake_tb'`, see command *2.5.39 Setting the wait time* for the motor movement.

Contained in firmware versions later than 15 March 2010.

config.SetBrakeTC

Declaration:

```
static native void SetBrakeTC(int time);
```

This function sets the wait time for switching off the motor voltage.

The motor current is switched off by resetting the enable input (see Section *2.5.25* "*Setting the function of* the digital inputs").

The function corresponds to the serial command `':brake_tc<time>'`, see command *2.5.40 Setting the wait time for* switching off the motor current.

Contained in firmware versions later than 15 March 2010.

config.GetBrakeTC

Declaration:

```
static native int GetBrakeTC();
```

This function reads out the wait time for switching off the motor voltage.

The motor current is switched off by resetting the enable input (see Section *2.5.25* "*Setting the function of* the digital inputs").

The function corresponds to the serial command `':brake_tc'`, see command *2.5.40 Setting the wait time for* switching off the motor current.

Contained in firmware versions later than 15 March 2010.

config.SetErrorCorrection

Declaration:

```
static native void SetErrorCorrection(int value);
```

This function sets the encoder monitoring mode.

The function corresponds to the serial command `'U<value>'`, see command *2.5.10 Setting the* error correction mode.

Contained in firmware versions later than 15 March 2010.

config.GetErrorCorrection

Declaration:

```
static native int GetErrorCorrection();
```

This function reads out the encoder monitoring mode.

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

Contained in firmware versions later than 15 March 2010.

config.SetSpeedmodeControl

Declaration:

```
static native void SetSpeedmodeControl(int value);
```

This function sets the control type for the speed mode.

The function corresponds to the serial command `':speedmode_control<value>'`, see command *2.9.3 Setting the control type* for the speed mode.

Contained in firmware versions later than 15 March 2010.

config.GetSpeedmodeControl

Declaration:

```
static native int GetSpeedmodeControl();
```

This function reads out the control type for the speed mode.

The function corresponds to the serial command `':speedmode_control'`, see command *2.9.3 Setting the control type* for the speed mode.

Contained in firmware versions later than 15 March 2010.

config.SetCLMotorType

Declaration:

```
static native void SetCLMotorType(int value);
```

This function defines the type of the connected motor.

The function corresponds to the serial command `':CL_motor_type<value>'`, see command *2.5.1 Setting the* motor type.

Contained in firmware versions later than 15 March 2010.

config.GetCLMotorType

Declaration:

```
static native int GetCLMotorType();
```

This function reads out the type of the connected motor.

The function corresponds to the serial command `':CL_motor_type'`, see command *2.5.1 Setting the* motor type.

Contained in firmware versions later than 15 March 2010.

config.SetFeedConstNum

Declaration:

```
static native void SetFeedConstNum(int value);
```

This function sets the numerator of the feed rate.

The function corresponds to the serial command `':feed_const_num<value>'`, see command *2.5.15 Setting the feed rate* numerator.

Contained in firmware versions later than 15 March 2010.

config.GetFeedConstNum

Declaration:

```
static native int GetFeedConstNum();
```

This function reads out the numerator of the feed rate.

The function corresponds to the serial command `':feed_const_num'`, see command *2.5.15 Setting the feed rate* numerator.

Contained in firmware versions later than 15 March 2010.

config.SetFeedConstDenum

Declaration:

```
static native void SetFeedConstDenum(int value);
```

This function sets the denominator of the feed rate.

The function corresponds to the serial command `':feed_const_denum<value>'`, see command *2.5.16 Setting the feed rate* denominator.

Contained in firmware versions later than 15 March 2010.

config.GetFeedConstDenum

Declaration:

```
static native int GetFeedConstDenum();
```

This function reads out the denominator of the feed rate.

The function corresponds to the serial command `':feed_const_denum'`, see command *2.5.16 Setting the feed rate* denominator.

Contained in firmware versions later than 15 March 2010.

config.SetCurrentTime

Declaration:

```
static native void SetCurrentTime(int time);
```

This function sets the current time constant for BLDC.

The function corresponds to the serial command `':itime<time>'`, see command *2.5.5 Setting the current time* constant for BLDC.

Contained in firmware versions later than 15 March 2010.

config.GetCurrentTime

Declaration:

```
static native int GetCurrentTime();
```

This function reads out the current time constant for BLDC.

The function corresponds to the serial command `':itime'`, see command *2.5.5 Setting the current time* constant for BLDC.

Contained in firmware versions later than 15 March 2010.

config.SetCurrentPeak

Declaration:

```
static native void SetCurrentPeak(int value);
```

This function sets the current peak value for BLDC.

The function corresponds to the serial command `':ipeak <value>'`, see command *2.5.4 Setting the peak current* for BLDC.

Contained in firmware versions later than 15 March 2010.

config.GetCurrentPeak

Declaration:

```
static native int GetCurrentPeak();
```

This function reads out the current peak value for BLDC.

The function corresponds to the serial command `':ipeak'`, see command *2.5.4 Setting the peak current* for BLDC.

Contained in firmware versions later than 15 March 2010.

config.ResetStartCount

Declaration:

```
static native void ResetStartCount(int value);
```

This function sets the switch-on counter.

The value can only have the value 1.

The function corresponds to the serial command `'%<value>'`, see command *2.7.7 Resetting switch-on* numerator.

Contained in firmware versions later than 15 March 2010.

config.GetStartCount

Declaration:

```
static native int GetStartCount();
```

This function reads out the switch-on counter.

The function corresponds to the serial command `'Z%'`, see command *2.7.7 Resetting switch-on* numerator.

Contained in firmware versions later than 15 March 2010.

config.SetLimitSwitchBehavior

Declaration:

```
static native void SetLimitSwitchBehavior(int value);
```

This function sets the limit switch behavior.

The value can only have the value 1.

The function corresponds to the serial command `'l<value>'`, see command *2.5.9 Setting the* limit switch behavior.

Contained in firmware versions later than 15 March 2010.

config.GetLimitSwitchBehavior

Declaration:

```
static native int GetLimitSwitchBehavior();
```

This function reads out the limit switch behavior.

The function corresponds to the serial command `'Zl'`, see command *2.5.9 Setting the* limit switch behavior.

Contained in firmware versions later than 15 March 2010.

config.SetMotorAddress

Declaration:

```
static native void SetMotorAddress(int value);
```

This function sets the motor address.

The function corresponds to the serial command `'m<value>'`, see command *2.5.7 Setting the* drive address.

Contained in firmware versions later than 15 March 2010.

config.GetMotorAddress

Declaration:

```
static native int GetMotorAddress();
```

This function reads the motor address.

The function corresponds to the serial command `'Zm'`, see command *2.5.7 Setting the* drive address.

Contained in firmware versions later than 15 March 2010.

3.5.5 “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 frequency in Hertz (steps per second).

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

The function corresponds to the serial command `'o<value>'`, see command *2.6.9 Setting the* maximum frequency.

drive.GetMaxSpeed

Declaration:

```
static native int GetMaxSpeed();
```

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

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

drive.SetMaxSpeed2

Declaration:

```
static native void SetMaxSpeed2(int speed);
```

Function sets the upper maximum frequency.

The function corresponds to the serial command `'n<value>'`, see command *2.6.10 Setting the maximum* frequency 2.

Contained in firmware versions later than 15 March 2010.

drive.GetMaxSpeed2

Declaration:

```
static native int GetMaxSpeed2();
```

Function reads out the upper maximum frequency.

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

Contained in firmware versions later than 15 March 2010.

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 *2.6.8 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 command *2.3 Read* command.

drive.SetAcceleration

Declaration:

```
static native void SetAcceleration(int value);
```

Specifies the acceleration ramp.

To convert the parameters 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 *2.6.11 Setting the* acceleration ramp.

drive.GetAcceleration

Declaration:

```
static native int GetAcceleration();
```

Reads out the currently valid value of the acceleration ramp.

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

drive.SetDeceleration

Declaration:

```
static native void SetDeceleration(int value);
```

This function sets the brake ramp.

The function corresponds to the serial command `'B<value>'`, see command *2.6.12 Setting the* brake ramp.

Contained in firmware versions later than 15 March 2010.

drive.GetDeceleration

Declaration:

```
static native int GetDeceleration();
```

This function reads out the brake ramp.

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

Contained in firmware versions later than 15 March 2010.

drive.SetDecelerationHalt

Declaration:

```
static native void SetDecelerationHalt(int value);
```

This function sets the quick stop ramp.

The function corresponds to the serial command `'H<value>'`, see command *2.6.13 Setting the* quickstop ramp.

Contained in firmware versions later than 15 March 2010.

drive.GetDecelerationHalt

Declaration:

```
static native int GetDecelerationHalt();
```

This function reads out the quick ramp.

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

Contained in firmware versions later than 15 March 2010.

drive.SetRampType

Declaration:

```
static native void SetRampType(int ramp);
```

This function sets the ramp type.

The function corresponds to the serial command `':ramp_mode<ramp>'`, see command *2.5.35 Setting the* ramp.

Contained in firmware versions later than 15 March 2010.

drive.GetRampType

Declaration:

```
static native int GetRampType();
```

This function reads out the ramp type.

The function corresponds to the serial command `':ramp_mode'`, see command *2.5.35 Setting the* ramp.

Contained in firmware versions later than 15 March 2010.

drive.SetJerk

Declaration:

```
static native void SetJerk(int value);
```

This function sets the maximum jerk for the acceleration in 100/s³.

The function corresponds to the serial command `':b<value>'`, see command *2.5.36 Setting the maximum jerk for* the acceleration ramp.

Contained in firmware versions later than 15 March 2010.

drive.GetJerk

Declaration:

```
static native int GetJerk();
```

This function outputs the maximum jerk for the acceleration in 100/s³.

The function corresponds to the serial command `'Z:b'`, see command *2.3 Read* command.

Contained in firmware versions later than 15 March 2010.

drive.SetBrakeJerk

Declaration:

```
static native void SetBrakeJerk(int value);
```

This function sets the brake jerk in 100/s³.

The function corresponds to the serial command `':B<value>'`, see command *2.5.37 Setting the maximum jerk for* the braking ramp.

Contained in firmware versions later than 15 March 2010.

drive.GetBrakeJerk

Declaration:

```
static native int GetBrakeJerk();
```

This function reads out the brake jerk in 100/s³.

The function corresponds to the serial command `'Z:B'`, see command *2.3 Read* command.

Contained in firmware versions later than 15 March 2010.

drive.IsReferenced

Declaration:

```
static native int IsReferenced();
```

This function reads out whether the motor is referenced or not.

The function corresponds to the serial command `':is_referenced'`, see command *2.5.21 Request "Motor is referenced"*.

Contained in firmware versions later than 15 March 2010.

drive.IncreaseFrequency

Declaration:

```
static native void IncreaseFrequency();
```

The function increases the speed in the speed mode by 100 steps/s.

The function corresponds to the serial command `'+'`, see command *2.7.9 Increasing the* speed.

Contained in firmware versions later than 15 March 2010.

drive.DecreaseFrequency

Declaration:

```
static native void DecreaseFrequency();
```

The function decreases the speed in the speed mode by 100 steps/s.

The function corresponds to the serial command `'+'`, see command *2.7.10 Reducing the* speed.

Contained in firmware versions later than 15 March 2010.

drive.TriggerOn

Declaration:

```
static native void TriggerOn();
```

Trigger for the flag positioning mode.

The function corresponds to the serial command `'T'`, see command *2.7.12 Actuating the* trigger.

Contained in firmware versions later than 15 March 2010.

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 *2.6.7 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 command *2.3 Read* command.

drive.SetMode

Declaration:

```
static native void SetMode(int value);
```

The positioning modes `'p'` are:

Positioning mode	
p=1	Relative positioning; The command *2.6.7 Setting the* travel distance `'s'` specifies the travel distance relative to the current position. The command *2.6.14 Setting the* direction of rotation `'d'` specifies the direction. The parameter *2.6.7 Setting the* travel distance `'s'` must be positive.
p=2	Absolute positioning; Command *2.6.7 Setting the* travel distance `'s'` defines the target position relative to the reference position. Command *2.6.14 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 *2.6.14 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 *2.5.9 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 *2.6.14 Setting the* direction of rotation `'d'` until it reaches the limit switch. Then a free run is performed, depending on the setting. See command *2.5.9 Setting the* limit switch behavior `'l'`.
Speed mode	
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	
p=6	Flag positioning mode; After starting, the motor runs up to the maximum speed. After arrival of the trigger event (command *2.7.12 Actuating the* trigger `'T'` or trigger input) the motor continues to travel the selected travel distance (command *2.6.7 Setting the* travel distance `'s'`) and changes its speed to the maximum speed 2 (command *2.6.10 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	
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	
p=11	Analog mode
Joystick mode	
p=12	Joystick mode
Analog positioning mode	
p=13	Analog positioning mode
p=3	Internal reference run; see Positioning mode
p=4	External reference run; see Positioning mode
HW reference mode	
p=14	HW reference mode
Torque mode	
p=15	Torque mode
CL quick test mode	
p=16	CL quick test mode
CL test mode	
p=17	CL test mode

drive.GetMode

Declaration:

```
static native int GetMode();
```

Reads out the current position type.

The function corresponds to the serial command `'Zp'`, see command *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 *2.5.2 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 command *2.3 Read* command.

drive.SetCurrentReduction

Declaration:

```
static native void SetCurrentReduction(int value);
```

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<value>'`, see command *2.5.3 Setting the phase current* at 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 command *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

The function corresponds to the serial command `'$'`, see command *2.5.22 Reading out* the status.

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 *2.6.14 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.SetDirectionReversing

Declaration:

```
static native void SetDirectionReversing (int value);
```

This function sets the reversal in the direction of rotation.

The function corresponds to the serial command `'t<value>'`, see command *2.6.15 Setting the* change of direction.

Contained in firmware versions later than 15 March 2010.

drive.GetDirectionReversing

Declaration:

```
static native int GetDirectionReversing ();
```

This function reads the value of the reversal in the direction of rotation.

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

Contained in firmware versions later than 15 March 2010.

drive.SetRepeat

Declaration:

```
static native void SetRepeat (int repeat);
```

This function sets the number of repetitions.

The function corresponds to the serial command `'W<repeat>'`, see command *2.6.16 Setting the* repetitions.

Contained in firmware versions later than 15 March 2010.

drive.GetRepeat

Declaration:

```
static native int GetRepeat ();
```

This function reads the number of repetitions.

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

Contained in firmware versions later than 15 March 2010.

drive.SetPause

Declaration:

```
static native void SetPause (int pause);
```

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

The function corresponds to the serial command `'P<pause>'`, see command *2.6.17 Setting the* record pause.

Contained in firmware versions later than 15 March 2010.

drive.GetPause

Declaration:

```
static native int GetPause ();
```

This function reads the pause time in milliseconds.

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

Contained in firmware versions later than 15 March 2010.

drive.SetNextRecord

Declaration:

```
static native void SetNextRecord (int record);
```

This function sets the next record.

The function corresponds to the serial command `'N< record>'`, see command *2.6.18 Setting the* continuation record.

Contained in firmware versions later than 15 March 2010.

drive.GetNextRecord

Declaration:

```
static native int GetNextRecord ();
```

This function reads out the number of the next record.

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

Contained in firmware versions later than 15 March 2010.

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 *2.5.19 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 *2.5.20 Reading out* the position.

drive.SetPosition

Declaration:

```
static native void SetPosition(int value);
```

Resets an error of the encoder monitor and sets the current and setpoint position to the value the parameter passes.

The function corresponds to the serial command `'D<value>'`, see command *2.5.17 Resetting the* position error.

Function contained in firmware versions later than 15 March 2010.

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.

The function corresponds to the serial command `'y'`, see command 2.6.3 Loading a record from the EEPROM.

drive.SaveDataSet

Declaration:

```
static native void SaveDataSet(int whichone);
```

Parameter: int whichone 1-32

Return: None

Writes the values in the controller memory to the selected data record.

The function corresponds to the serial command `'>'`, see command *2.6.5 . Saving a* record*.*

Function contained in firmware versions later than 15 March 2010.

3.5.6 "dspdrive" class

Application

The dspdrive class is used to configure the current controller in controllers which are equipped with a DSP drive.

dspdrive.SetDSPDrivePLow

Declaration:

```
static native void SetDSPDrivePLow(int value);
```

This function sets the P component of the current controller at a standstill.

The function corresponds to the serial command `':dspdrive_KP_low<value>'`, see command *2.12.1 Setting the P component of the* current controller at standstill.

Contained in firmware versions later than 15 March 2010.

dspdrive.GetDSPDrivePLow

Declaration:

```
static native int GetDSPDrivePLow();
```

This function reads out the P component of the current controller at a standstill.

The function corresponds to the serial command `':dspdrive_KP_low'`, see command *2.12.1 Setting the P component of the* current controller at standstill.

Contained in firmware versions later than 15 March 2010.

dspdrive.SetDSPDrivePHigh

Declaration:

```
static native void SetDSPDrivePHigh(int value);
```

This function sets the P component of the current controller during the run.

The function corresponds to the serial command `':dspdrive_KP_hig<value>'`, see command *2.12.2 Setting the P component of the current* controller during the run.

Contained in firmware versions later than 15 March 2010.

dspdrive.GetDSPDrivePHigh

Declaration:

```
static native int GetDSPDrivePHigh();
```

This function reads out the P component of the current controller during the run.

The function corresponds to the serial command `':dspdrive_KP_hig'`, see command *2.12.2 Setting the P component of the current* controller during the run.

Contained in firmware versions later than 15 March 2010.

dspdrive.SetDSPDrivePScale

Declaration:

```
static native void SetDSPDrivePScale(int value);
```

This function sets the scaling factor to speed-independent. adjustment of the P component of the controller during the run.

The function corresponds to the serial command `':dspdrive_KP_scale<value>'`, see command *2.12.3 Setting the scaling factor for speed-dependent adjustment of the P component of* the controller during the run.

Contained in firmware versions later than 15 March 2010.

dspdrive.GetDSPDrivePScale

Declaration:

```
static native int GetDSPDrivePScale();
```

The function reads out the scaling factor for the speed-dependent adjustment of the P component of the controller during the run.

The function corresponds to the serial command `':dspdrive_KP_scale'`, see command *2.12.3 Setting the scaling factor for speed-dependent adjustment of the P component of* the controller during the run.

Contained in firmware versions later than 15 March 2010.

dspdrive.SetDSPDriveILow

Declaration:

```
static native void SetDSPDriveILow(int value);
```

This function sets the I component of the current controller at a standstill.

The function corresponds to the serial command `':dspdrive_KI_low<value>'`, see command *2.12.4 Setting the I component of the* current controller at standstill.

Contained in firmware versions later than 15 March 2010.

dspdrive.GetDSPDriveILow

Declaration:

```
static native int GetDSPDriveILow();
```

This function reads out the I component of the current controller at a standstill.

The function corresponds to the serial command `':dspdrive_KP_low'`, see command *2.12.4 Setting the I component of the* current controller at standstill.

Contained in firmware versions later than 15 March 2010.

dspdrive.SetDSPDriveIHigh

Declaration:

```
static native void SetDSPDriveIHigh(int value);
```

This function sets the I component of the current controller during the run.

The function corresponds to the serial command `':dspdrive_KI_hig<value>'`, see command *2.12.5 Setting the I component of the current* controller during the run.

Contained in firmware versions later than 15 March 2010.

dspdrive.GetDSPDriveIHigh

Declaration:

```
static native int GetDSPDriveIHigh();
```

This function reads out the I component of the current controller during the run.

The function corresponds to the serial command `':dspdrive_KI_hig'`, see command *2.12.5 Setting the I component of the current* controller during the run.

Contained in firmware versions later than 15 March 2010.

dspdrive.SetDSPDriveIScale

Declaration:

```
static native void SetDSPDriveIScale(int value);
```

This function sets the scaling factor for the speed-dependent adjustment of the I component of the controller during the run.

The function corresponds to the serial command `':dspdrive_KI_scale<value>'`, see command *2.12.6 Setting the scaling factor for speed-dependent adjustment of the I component of* the controller during the run.

Contained in firmware versions later than 15 March 2010.

dspdrive.GetDSPDriveIScale

Declaration:

```
static native int GetDSPDriveIScale();
```

This function reads out the scaling factor for the speed-dependent adjustment of the I component of the controller during the run.

The function corresponds to the serial command `':dspdrive_KI_scale'`, see command *2.12.6 Setting the scaling factor for speed-dependent adjustment of the I component of* the controller during the run.

Contained in firmware versions later than 15 March 2010.

3.5.7 “io” class

Application

The io class is used to manage the digital and analog inputs and outputs.

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).

io.SetAnalogDead

Declaration:

```
static native void SetAnalogDead(int analogDead);
```

This function sets the dead range of the analog input.

The function corresponds to the serial command `'=<value>'`, see command *2.7.1 Setting the dead* range for the joystick mode.

Contained in firmware versions later than 15 March 2010.

io.GetAnalogDead

Declaration:

```
static native int GetAnalogDead();
```

This function reads out the dead range of the analog input.

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

Contained in firmware versions later than 15 March 2010.

io.SetAnalogFilter

Declaration:

```
static native void SetAnalogFilter(int filter);
```

This function sets the value for the filter of the analog input.

The function corresponds to the serial command `'f<filter>'`, see command *2.7.2 Setting the filter for the analog* and joystick modes.

Contained in firmware versions later than 15 March 2010.

io.GetAnalogFilter

Declaration:

```
static native int GetAnalogFilter();
```

This function reads out the value for the filter of the analog input.

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

Contained in firmware versions later than 15 March 2010.

io.SetInputMaskEdge

Declaration:

```
static native void SetInputMaskEdge(int mask);
```

This function sets the polarity of the inputs and outputs.

The function corresponds to the serial command `'h<mask>'`, see command *2.5.27 Reversing the polarity of the inputs* and outputs.

Contained in firmware versions later than 15 March 2010.

io.GetInputMaskEdge

Declaration:

```
static native int GetInputMaskEdge();
```

This function reads out the current polarity of the inputs and outputs.

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

Contained in firmware versions later than 15 March 2010.

io.SetDebounceTime

Declaration:

```
static native void SetDebounceTime(int time);
```

This function sets the debounce time for the inputs in milliseconds.

The function corresponds to the serial command `'K<time>'`, see command *2.5.28 Setting the debounce time for* the inputs.

Contained in firmware versions later than 15 March 2010.

io.GetDebounceTime

Declaration:

```
static native int GetDebounceTime();
```

This function reads out the debounce time for the inputs in milliseconds.

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

Contained in firmware versions later than 15 March 2010.

io.SetInput1Selection

Declaration:

```
static native void SetInput1Selection(int function);
```

This function sets the function for digital input 1.

The function corresponds to the serial command `':port_in_a<function>'`, see command *2.5.25 Setting the function of* the digital inputs.

Contained in firmware versions later than 15 March 2010.

io.GetInput1Selection

Declaration:

```
static native int GetInput1Selection();
```

This function reads out the function for digital input 1.

The function corresponds to the serial command `':port_in_a'`, see command *2.5.25 Setting the function of* the digital inputs.

Contained in firmware versions later than 15 March 2010.

io.SetInput2Selection

Declaration:

```
static native void SetInput2Selection(int function);
```

This function sets the function for digital input 2.

The function corresponds to the serial command `':port_in_b<function>'`, see command *2.5.25 Setting the function of* the digital inputs.

Contained in firmware versions later than 15 March 2010.

io.GetInput2Selection

Declaration:

```
static native int GetInput2Selection();
```

This function reads out the function for digital input 2.

The function corresponds to the serial command `':port_in_b'`, see command *2.5.25 Setting the function of* the digital inputs.

Contained in firmware versions later than 15 March 2010.

io.SetInput3Selection

Declaration:

```
static native void SetInput3Selection(int function);
```

This function sets the function for digital input 3.

The function corresponds to the serial command `':port_in_c<function>'`, see command *2.5.25 Setting the function of* the digital inputs.

Contained in firmware versions later than 15 March 2010.

io.GetInput3Selection

Declaration:

```
static native int GetInput3Selection();
```

This function reads out the function for digital input 3.

The function corresponds to the serial command `':port_in_c'`, see command *2.5.25 Setting the function of* the digital inputs.

Contained in firmware versions later than 15 March 2010.

io.SetInput4Selection

Declaration:

```
static native void SetInput4Selection(int function);
```

This function sets the function for digital input 4.

The function corresponds to the serial command `':port_in_d<function>'`, see command *2.5.25 Setting the function of* the digital inputs.

Contained in firmware versions later than 15 March 2010.

io.GetInput4Selection

Declaration:

```
static native int GetInput4Selection();
```

This function reads out the function for digital input 4.

The function corresponds to the serial command `':port_in_d'`, see command *2.5.25 Setting the function of* the digital inputs.

Contained in firmware versions later than 15 March 2010.

io.SetInput5Selection

Declaration:

```
static native void SetInput5Selection(int function);
```

This function sets the function for digital input 5.

The function corresponds to the serial command `':port_in_e<function>'`, see command *2.5.25 Setting the function of* the digital inputs.

Contained in firmware versions later than 15 March 2010.

io.GetInput5Selection

Declaration:

```
static native int GetInput5Selection();
```

This function reads out the function for digital input 5.

The function corresponds to the serial command `':port_in_e'`, see command *2.5.25 Setting the function of* the digital inputs.

Contained in firmware versions later than 15 March 2010.

io.SetInput6Selection

Declaration:

```
static native void SetInput6Selection(int function);
```

This function sets the function for digital input 6.

The function corresponds to the serial command `':port_in_f<function>'`, see command *2.5.25 Setting the function of* the digital inputs.

Contained in firmware versions later than 15 March 2010.

io.GetInput6Selection

Declaration:

```
static native int GetInput6Selection();
```

This function reads out the function for digital input 6.

The function corresponds to the serial command `':port_in_f'`, see command *2.5.25 Setting the function of* the digital inputs.

Contained in firmware versions later than 15 March 2010.

io.SetInput7Selection

Declaration:

```
static native void SetInput7Selection(int function);
```

This function sets the function for digital input 7.

The function corresponds to the serial command ':port_in_g<function>', see command *2.5.25 Setting the function of* the digital inputs.

Contained in firmware versions later than 15 March 2010.

io.GetInput7Selection

Declaration:

```
static native int GetInput7Selection();
```

This function reads out the function for digital input 7.

The function corresponds to the serial command ':port_in_g', see command *2.5.25 Setting the function of* the digital inputs.

Contained in firmware versions later than 15 March 2010.

io.SetInput8Selection

Declaration:

```
static native void SetInput8Selection(int function);
```

This function sets the function for digital input 8.

The function corresponds to the serial command ':port_in_h<function>', see command *2.5.25 Setting the function of* the digital inputs.

Contained in firmware versions later than 15 March 2010.

io.GetInput8Selection

Declaration:

```
static native int GetInput8Selection();
```

This function reads out the function for digital input 8.

The function corresponds to the serial command ':port_in_h', see command *2.5.25 Setting the function of* the digital inputs.

Contained in firmware versions later than 15 March 2010.

io.SetOutput1Selection

Declaration:

```
static native void SetOutput1Selection(int function);
```

This function sets the function for digital output 1.

The function corresponds to the serial command ':port_out_a<function>', see command *2.5.26 Setting the function of* the digital outputs.

Contained in firmware versions later than 15 March 2010.

io.GetOutput1Selection

Declaration:

```
static native int GetOutput1Selection();
```

This function reads out the function for digital output 1.

The function corresponds to the serial command `':port_out_a'`, see command *2.5.26 Setting the function of* the digital outputs.

Contained in firmware versions later than 15 March 2010.

io.SetOutput2Selection

Declaration:

```
static native void SetOutput2Selection(int function);
```

This function sets the function for digital output 2.

The function corresponds to the serial command `':port_out_b<function>'`, see command *2.5.26 Setting the function of* the digital outputs.

Contained in firmware versions later than 15 March 2010.

io.GetOutput2Selection

Declaration:

```
static native int GetOutput2Selection();
```

This function reads out the function for digital output 2.

The function corresponds to the serial command `':port_out_b'`, see command *2.5.26 Setting the function of* the digital outputs.

Contained in firmware versions later than 15 March 2010.

io.SetOutput3Selection

Declaration:

```
static native void SetOutput3Selection(int function);
```

This function sets the function for digital output 3.

The function corresponds to the serial command `':port_out_c<function>'`, see command *2.5.26 Setting the function of* the digital outputs.

Contained in firmware versions later than 15 March 2010.

io.GetOutput3Selection

Declaration:

```
static native int GetOutput3Selection();
```

This function reads out the function for digital output 3.

The function corresponds to the serial command `':port_out_c'`, see command *2.5.26 Setting the function of* the digital outputs.

Contained in firmware versions later than 15 March 2010.

io.SetOutput4Selection

Declaration:

```
static native void SetOutput4Selection(int function);
```

This function sets the function for digital output 4.

The function corresponds to the serial command `':port_out_d<function>'`, see command *2.5.26 Setting the function of* the digital outputs.

Contained in firmware versions later than 15 March 2010.

io.GetOutput4Selection

Declaration:

```
static native int GetOutput4Selection();
```

This function reads out the function for digital output 4.

The function corresponds to the serial command `':port_out_d'`, see command *2.5.26 Setting the function of* the digital outputs.

Contained in firmware versions later than 15 March 2010.

io.SetOutput5Selection

Declaration:

```
static native void SetOutput5Selection(int function);
```

This function sets the function for digital output 5.

The function corresponds to the serial command `':port_out_e<function>'`, see command *2.5.26 Setting the function of* the digital outputs.

Contained in firmware versions later than 15 March 2010.

io.GetOutput5Selection

Declaration:

```
static native int GetOutput5Selection();
```

This function reads out the function for digital output 5.

The function corresponds to the serial command `':port_out_e'`, see command *2.5.26 Setting the function of* the digital outputs.

Contained in firmware versions later than 15 March 2010.

io.SetOutput6Selection

Declaration:

```
static native void SetOutput6Selection(int function);
```

This function sets the function for digital output 6.

The function corresponds to the serial command `':port_out_f<function>'`, see command *2.5.26 Setting the function of* the digital outputs.

Contained in firmware versions later than 15 March 2010.

io.GetOutput6Selection

Declaration:

```
static native int GetOutput6Selection();
```

This function reads out the function for digital output 6.

The function corresponds to the serial command `':port_out_f'`, see command *2.5.26 Setting the function of* the digital outputs.

Contained in firmware versions later than 15 March 2010.

io.SetOutput7Selection

Declaration:

```
static native void SetOutput7Selection(int function);
```

This function sets the function for digital output 7.

The function corresponds to the serial command `':port_out_g<function>'`, see command *2.5.26 Setting the function of* the digital outputs.

Contained in firmware versions later than 15 March 2010.

io.GetOutput7Selection

Declaration:

```
static native int GetOutput7Selection();
```

This function reads out the function for digital output 7.

The function corresponds to the serial command `':port_out_g'`, see command *2.5.26 Setting the function of* the digital outputs.

Contained in firmware versions later than 15 March 2010.

io.SetOutput8Selection

Declaration:

```
static native void SetOutput8Selection(int function);
```

This function sets the function for digital output 8.

The function corresponds to the serial command `':port_out_h<function>'`, see command *2.5.26 Setting the function of* the digital outputs.

Contained in firmware versions later than 15 March 2010.

io.GetOutput8Selection

Declaration:

```
static native int GetOutput8Selection();
```

This function reads out the function for digital output 8.

The function corresponds to the serial command `':port_out_h'`, see command *2.5.26 Setting the function of* the digital outputs.

Contained in firmware versions later than 15 March 2010.

io.SetAnalogMin

Declaration:

```
static native void SetAnalogMin(int value);
```

This function sets the minimum voltage for the analog input.

The function corresponds to the serial command `'Q<value>'`, see command *2.7.3 Setting the minimum voltage* for the analog mode.

Contained in firmware versions later than 15 March 2010.

io.GetAnalogMin

Declaration:

```
static native int GetAnalogMin();
```

This function reads out the minimum voltage for the analog input.

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

Contained in firmware versions later than 15 March 2010.

io.SetAnalogMax

Declaration:

```
static native void SetAnalogMax(int value);
```

This function sets the maximal voltage for the analog input.

The function corresponds to the serial command `'R<value>'`, see command *2.7.4 Setting the maximum voltage* for the analog mode.

Contained in firmware versions later than 15 March 2010.

io.GetAnalogMax

Declaration:

```
static native int GetAnalogMax();
```

This function reads out the maximum voltage for the analog input.

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

Contained in firmware versions later than 15 March 2010.

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

util.SetBit

Declaration:

```
static int SetBit(int value, int whichone);
```

Sets a bit in an integer.

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

util.SetStepMode

Declaration:

```
static native void SetStepMode(int value);
```

This function sets the step mode.

The function corresponds to the serial command `'g<value>'`, see command *2.5.6 Setting the* step mode.

Contained in firmware versions later than 15 March 2010.

util.GetStepMode

Declaration:

```
static native int GetStepMode();
```

This function reads out the step mode.

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

Contained in firmware versions later than 15 March 2010.

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 moves to a
 *  position calculated from it
 *
 * */
import nanotec.*;
class AnalogExample {
       /** Reads out 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 the motor
              util.SetStepMode(4);         //1/4 step
              drive.SetTargetPos(0);       //Target position:0
              drive.SetMaxSpeed(2000);     //Speed
              drive.SetMode(2);            //Absolute positioning
              //Main loop
              while(true){
                     io.SetLED(1);
                     util.Sleep(100);

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

                     drive.StopDrive( 0 );
                     drive.SetTargetPos( CalculateTargetPos () );
                     drive.StartDrive( );
              }
       }
}
```

3.6.2 DigitalExample.java

```
/** When input 1 is active, the LED is switched on
 *  When input 2 is active, the value of the analog input is sent via the
 *  serial interface
 *
 *
 * */

import nanotec.*;

class DigitalExample {

       public static void main() {

              int input = 0;
              int cnt = 0;

              //Main loop
              while(true){

                     input =  io.GetDigitalInput( );

                     //Bit 0 corresponds to input 1
                     if( util.TestBit(input,0) ){
                            io.SetLED(1);
                     } else {
                            io.SetLED(0);
                     }


                     cnt ++;
                     //Do not send analog value permanently since //hard to
                     read
                     if( util.TestBit(input,1) && ((cnt % 50) == 0)){
                            comm.SendInt( io.GetAnalogInput(1) );
                     }
              }
       }
}
```

3.6.3 TimerExample.java

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

import nanotec.*;

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

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

import nanotec.*;

class ConfigDriveExample {

       public static void main() {

              //Configure the motor
              drive.SetMode(2);                    //Absolute positioning
              drive.SetMinSpeed(100);
              drive.SetAcceleration(2000);         //Ramp
              drive.SetCurrent(10);                //Current
              drive.SetCurrentReduction(1);        //Current for reduction
              util.SetStepMode(2);                 //1/2 step mode


              //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.*;

class DigitalOutput {

	public static void main() {

		util.Sleep(200);

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

			io.SetDigitalOutput(2);
			comm.SendInt( io.GetDigitalOutput( ) );
			util.Sleep(1000);

			io.SetDigitalOutput(4);
			comm.SendInt( io.GetDigitalOutput( ) );
			util.Sleep(1000);

			io.SetDigitalOutput(7);
			comm.SendInt( io.GetDigitalOutput( ) );
			util.Sleep(1000);

			io.SetDigitalOutput(0);
			comm.SendInt( io.GetDigitalOutput( ) );
			util.Sleep(1000);
		}
	}
}
```

3.6.6 ExportAnalogIn.java

```
/** Reads the analog value and scales it. The result
 * is written  to the "Joystick mode dead range" setting.
 * In this way, the current value can be read out with the 'Z='
 * command (e.g. #1Z= for motor ID 1)
 * Please note: Since the setting for the dead range is changed,
 * this program cannot be operated together with an analog
 * mode.
*/

import nanotec.*;

class ExportAnalogIn {

        public static void main() {
                while(true){
                        util.Sleep(1000);
                        io.SetAnalogDead((io.GetAnalogInput(1) - 500) / 10);
                }
        }

}
```

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. However, it is recommended that you use NanoJEasy since it is more comfortable to use and less fault-prone.

Java SE

NanoJEasy contains the free Java compiler gcj of the GNU project to translate Java files. It is located within the NanoJEasy installation directory in the java/bin directory.

Alternatively, the standard Java implementation Java SE from Oracle can also be used. The JDK (Java Development Kit) can be downloaded free of charge from the oracle.com website.

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.
4	Click on <Edit> under the “System variables” window.
5	Enter the NanoJEasy installation path under “Value of the variables”.
6	Click on <OK>.

Firmware utility

The firmware 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 firmware_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 GNU Java compiler:

```
gcj.exe –C Meinprogramm.java
```

Alternatively, the program can be translated with the normal Java SE compiler:

```
javac.exe Meinprogramm.java
```

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

```
Meinprogramm.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

Enter:

```
ejvm_linker.exe Meinprogramm.class Meinprogramm.prg
```

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

Usually, the Nanotec classes that can be linked need to be additionally specified:

```
ejvm_linker.exe Meinprogramm.class nanotec\comm.class
nanotec\config.class nanotec\drive.class nanotec\io.class
nanotec\cl.class nanotec\util.class nanotec\dspdrive.class
nanotec\capture.class Meinprogramm.prg
```

Result

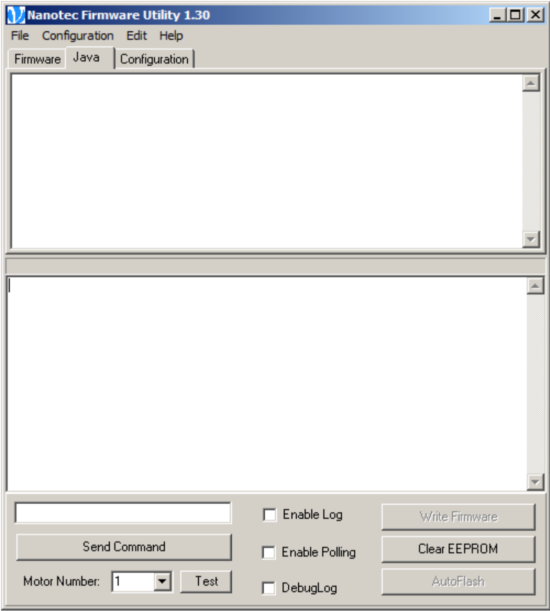
The result of the linking and conversion is a .prg file which can be loaded into the controller:

```
Meinprogramm.prg
```

3.7.4 Transferring the program to the controller

Firmware utility dialog window

The transfer to the controller is performed using the firmware 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 of the firmware utility is filled out.
4	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 firmware 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:

`(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 *2.8.4 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 *2.8.5 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 *2.8.6 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).
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).

Index	Error message	Meaning
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 *2.8.5 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 1.60.0.0 and SDK-version 0.60.0.0.

Prerequisites

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.

Programming environments

Microsoft Visual Studio or any other suitable high language IDE can be used as the programming environment. The sample projects delivered with NanoPro were created with Microsoft Visual Studio.

Programming examples

Several examples for the use of CommandsPD41 are provided in the NanoPro installation directory in the SDK\example subdirectory. All examples are implemented as projects for Microsoft Visual Studio.

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

4.3.1 Generic

Methods

There are two categories of methods:

- Set methods which pass information to the controller. The value returned in the 'Set' method can be used to check that the information has also been sent to the controller.
- Get methods that fetch information from 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 return

4.3.2 List of functions

ErrorFlag

Definition:

```
bool ErrorFlag
```

If this variable has the value true, an error occurred.

ErrorNumber

Definition:

```
int ErrorNumber
```

If an error occurred, this variable stores the number of the error.

ErrorMessageString

Definition:

```
string ErrorMessageString
```

If an error occurred, this variable stores the description of the error.

SerialPorts

Definition:

```
string[] SerialPorts
```

This field contains a list of available serial interfaces of the computer system.

SelectedPort

Definition:

```
string SelectedPort
```

This variable is used to define the serial interface to be used (e.g. "COM1").

Baudrate

Definition:

```
int Baudrate
```

This variable is used to define the transmission rate to be used.

Supportlog

Definition:

```
bool Supportlog
```

This variable is used to define whether a support log should be written.

GetAvailableMotorAddresses

Definition:

```
IList<int> GetAvailableMotorAddresses
```

This field contains a list of possible motor addresses.

MotorAddresse

Definition:

```
int MotorAddresse
```

This variable defines the motor addresses to be used for communication.

GetStatusByte

Definition:

```
byte GetStatusByte()
```

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

The function corresponds to the serial command `'$'`.

IsMotorReady

Definition:

```
bool IsMotorReady()
```

This function returns true if bit 0 in the status byte is set (controller is ready).

IsAtReferencePosition

Definition:

```
bool IsAtReferencePosition()
```

This function returns true if bit 1 in the status byte is set (zero position reached).

HasPositionError

Definition:

```
bool HasPositionError()
```

This function returns true if bit 2 in the status byte is set (position error).

HasEndedTravelProfileAndStartInputStillActive

Definition:

```
bool HasEndedTravelProfileAndStartInputStillActive()
```

This function returns true if bit 3 in the status byte is set (input 1 is set while controller is ready again).

IsPositionModeActive

Definition:

```
bool IsPositionModeActive()
```

This function returns true if the positioning mode is active.

IsSpeedModeActive

Definition:

```
bool IsSpeedModeActive()
```

This function returns true if the speed mode is active.

IsFlagPositionModeActive

Definition:

```
bool IsFlagPositionModeActive()
```

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

IsClockDirectionModeActive

Definition:

```
bool IsClockDirectionModeActive()
```

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

IsJoyStickModeActive

Definition:

```
bool IsJoyStickModeActive()
```

This function returns true if the joystick mode is active.

IsAnalogModeActive

Definition:

```
bool IsAnalogModeActive()
```

This function returns true if the analog mode is active.

IsTorqueModeActive

Definition:

```
bool IsTorqueModeActive()
```

This function returns true if the torque mode is active.

IsMasterModeActive

Definition:

```
bool IsMasterModeActive()
```

This function returns true if the master mode ("!10") is active.

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 serial command `'A'`.

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 serial command `'S1'`.

QuickStopTravelProfile

Definition:

```
bool  QuickStopTravelProfile()
```

This function can be used to stop the travel profile rapidly.

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

The function corresponds to serial command `'S'`.

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 serial command `'+'`.

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 serial command `'-'`.

TriggerOn

Definition:

```
bool TriggerOn()
```

This function sends the trigger command to 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 serial command `'T'`.

SetRamp

Definition:

```
bool SetRamp(int ramp)
```

This function sets the acceleration 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 serial command `'b'`.

SetBreak

Definition:

```
bool SetBreak(double breakTime)
```

This function sets the pause time 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 serial command `'P'`.

ChooseRecord

Definition:

```
bool ChooseRecord(int recordNumber)
```

This function loads a specific record (travel profile).

The recordNumber parameter is the record number (travel profile) that should be loaded.

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

The function corresponds to serial command `'y'`.

GetRamp

Definition:

```
int GetRamp(int operationNumber)
```

This function reads out the acceleration ramp.

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

The function corresponds to serial command `'Zb'`.

GetBreak

Definition:

```
int GetBreak(int operationNumber)
```

This function reads the pause time in milliseconds.

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

The function corresponds to serial command `'ZP'`.

SetDirection

Definition:

```
bool SetDirection(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 corresponds to serial command `'d'`.

SetMaxFrequency

Definition:

```
bool SetMaxFrequency(int maxFrequency)
```

This function sets the target frequency.

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

The function corresponds to serial command `'o'`.

GetMaxFrequency

Definition:

```
int GetMaxFrequency(int operationNumber)
```

This function reads out the target frequency.

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

The function corresponds to serial command `'Zo'`.

SetRotationMode

Definition:

```
bool SetRotationMode(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 during travel

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

The "Check during travel" setting exists for compatibility reasons and is equivalent to the "Check at end" behavior. To actually make a correction during travel, the closed loop mode should be used.

The function corresponds to serial command `'U'`.

ResetPositionError

Definition:

```
bool ResetPositionError(bool useEncoderValue, int Position)
```

This function can be used to reset a position error and set the value of the position counter.

- useEncoderValue = true: set position counter to value displayed by the encoder
- useEncoderValue = false: set position counter to the value of the position variable

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

The function corresponds to serial command `'D'`.

ResetAllSettings

Definition:

```
bool ResetAllSettings()
```

This function sets all settings of the controller back to default values (factory default settings).

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

The function corresponds to serial command `'~'`.

GetVersion

Definition:

```
string GetVersion()
```

This function returns the version string of the controller.

The function corresponds to serial command `'v'`.

SetSendStatusWhenCompleted

Definition:

```
bool SetSendStatusWhenCompleted(bool sendStatus)
```

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

- sendStatus = 0: automatic sending off
- sendStatus = 1: automatic sending 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 serial command `'J'`.

GetSendStatusWhenCompleted

Definition:

```
bool GetSendStatusWhenCompleted()
```

This function reads whether the independent sending of a status at the end of a run is switched on.

- sendStatus = 0: automatic sending off
- sendStatus = 1: automatic sending on

The function corresponds to serial command `'ZJ'`.

GetPosition

Definition:

```
int GetPosition()
```

This function outputs the value of the position counter.

The function corresponds to serial command `'C'`.

GetIO

Definition:

```
int GetIO()
```

This function returns the status of the inputs as an integer value.

The function corresponds to serial command `'ZY'`.

SetIO

Definition:

```
bool SetIO(int io)
```

This function sets the status of the outputs via an integer value.

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

The function corresponds to serial command `'Y'`.

SetInputMaskEdge

Definition:

```
bool SetInputMaskEdge(int ioMask)
```

This function sets the polarity of the inputs and outputs.

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

For an exact description of the use, see serial command `'h'`.

GetInputMaskEdge

Definition:

```
int GetInputMaskEdge()
```

This function outputs the current polarity of the inputs and outputs.

The function corresponds to serial command `'Zh'`.

SetRecord

Definition:

```
bool SetRecord(int recordNumber)
```

This function saves the record parameters previously set in the record with the number passed in the parameter.

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

The function corresponds to serial command `'>'`.

SetPlay

Definition:

```
bool SetPlay(int play)
```

This function sets the dead range of the analog input.

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

The function corresponds to serial command `'='`.

GetPlay

Definition:

```
int GetPlay()
```

This function returns the value of the dead range of the analog input.

The function corresponds to serial command `'Z='`.

SetDebounceTime

Definition:

```
bool SetDebounceTime(int debounceTime)
```

This function sets the debounce time for the inputs 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 serial command `'K'`.

GetDebounceTime

Definition:

```
int GetDebounceTime()
```

This function returns the debounce time for the inputs in milliseconds.

The function corresponds to serial command `'ZK'`.

SetSoftwareFilter

Definition:

```
bool SetSoftwareFilter(int softwareFilter)
```

This function sets the value for the filter of the analog input.

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

The function corresponds to serial command `'f'`.

GetSoftwareFilter

Definition:

```
int GetSoftwareFilter()
```

This function reads out the value for the filter of the analog input.

The function corresponds to serial command `'Zf'`.

SetStepMode

Definition:

```
bool SetStepMode(int stepMode)
```

This function sets the step mode.

- stepMode = 1 corresponds to a full step
- stepMode = 2 corresponds to half of a step
- stepMode = 4 corresponds to a quarter of a step
- stepMode = 5 corresponds to a fifth of a step
- stepMode = 8 corresponds to an eighth of a step
- stepMode = 10 corresponds to a tenth of a step
- stepMode = 16 corresponds to a 16th of a step
- stepMode = 32 corresponds to a 32nd of a step

- stepMode = 64 corresponds to a 64th of a step
- stepMode = 254 corresponds to the feed rate
- stepMode = 255 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 serial command `'g'`.

GetStepMode

Definition:

```
int GetStepMode()
```

This function reads out the current step mode.

- Return = 1 corresponds to full step
- Return = 2 corresponds to half of a step
- Return = 4 corresponds to a quarter of a step
- Return = 5 corresponds to a fifth of a step
- Return = 8 corresponds to an eighth of a step
- Return = 10 corresponds to a tenth of a step
- Return = 16 corresponds to a 16th of a step
- Return = 32 corresponds to a 32nd of a step
- Return = 64 corresponds to a 64th of a step
- Return = 254 corresponds to the feed rate
- Return = 255 corresponds to adaptive microstep

The function corresponds to serial command `'Zg'`.

SetMotorAddress

Definition:

```
bool SetMotorAddress(int newMotorAddress)
```

This function sets the 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 serial command `'m'`.

GetMotorAddress

Definition:

```
int GetMotorAddress(int selectedMotor)
```

This function reads out the motor address. The value of the passed parameter selectedMotor is irrelevant since the command is sent to all bus users.

Attention:
When this command is used, only one controller should be connected to the RS485 bus.

GetErrorAddress

Definition:

```
int GetErrorAddress()
```

This function reads the error address at which the last error code is found.

The function corresponds to serial command `'E'`.

GetError

Definition:

```
int GetError(int errorAddress)
```

This function reads the error (status) to the address handed over.

The function corresponds to serial command `'ZE'`.

SetEnableAutoCorrect

Definition:

```
bool SetEnableAutoCorrect(string recordNumber, bool
autoCorrect)
```

This function configures on the automatic error correction of the motor.

The value of autoCorrect specifies whether a correction should take place.

The recordNumber parameter is the record number (travel profile) with which an error should be corrected.

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

The function corresponds to serial command `'F'`.

GetEnableAutoCorrect

Definition:

```
int GetEnableAutoCorrect(int errorAddress)
```

This function reads out which record is set for the automatic error correction.

The function corresponds to serial command `'ZF'`.

SetSwingOutTime

Definition:

```
bool SetSwingOutTime(int swingOutTime)
```

This function sets the swing out time.

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

The function corresponds to serial command `'O'`.

GetSwingOutTime

Definition:

```
int GetSwingOutTime()
```

This function reads out the swing out time.

The function corresponds to serial command `'ZO'`.

SetNextOperation

Definition:

```
bool SetNextOperation(int operationNumber)
```

This function sets the next record.

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

The function corresponds to serial command `'N'`.

GetNextOperation

Definition:

```
int GetNextOperation(int operationNumber)
```

This function reads out the number of the next record.

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

The function corresponds to serial command `'ZN'`.

SetPhaseCurrent

Definition:

```
bool SetPhaseCurrent(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 serial command `'i'`.

GetPhaseCurrent

Definition:

```
int GetPhaseCurrent()
```

This function returns the phase current in percent.

The function corresponds to serial command `'Zi'`.

SetCurrentReduction

Definition:

```
bool SetCurrentReduction(int currentReduction)
```

This function sets the phase current at a standstill 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 serial command `'r'`.

GetCurrentReduction

Definition:

```
int GetCurrentReduction()
```

This function returns the phase current at a standstill in percent.

The function corresponds to serial command `'Zr'`.

SetLimitSwitchBehavior

Definition:

```
bool SetLimitSwitchBehavior(int refBehaviorsInternal, int
norBehaviorsInternal, int refBehaviorsExternal, int
norBehaviorsExternal)
```

This function sets the limit switch behavior.

The individual parameters have the following meanings:

- refBehaviorsInternal = behavior of the internal limit switch during a reference run
- norBehaviorsInternal = behavior of the internal limit switch during a normal run
- refBehaviorsExternal = behavior of the external limit switch during a reference run
- norBehaviorsExternal = behavior of the external limit switch during a normal run

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

For an exact description of the use, see serial command `'l'`.

GetLimitSwitchBehavior

Definition:

```
bool GetLimitSwitchBehavior(out int refBehaviorsInternal,
out int norBehaviorsInternal, out int
refBehaviorsExternal, out int norBehaviorsExternal)
```

This function reads out the limit switch behavior.

The individual return parameters have the following meanings:

- refBehaviorsInternal = behavior of the internal limit switch during a reference run
- norBehaviorsInternal = behavior of the internal limit switch during a normal run
- refBehaviorsExternal = behavior of the external limit switch during a reference run
- norBehaviorsExternal = behavior of the external limit switch during a normal run

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

For an exact description of the use, see serial command `'l'`.

SetReverseClearance

Definition:

```
bool SetReverseClearance(int reverseClearance)
```

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 serial command `'z'`.

GetReverseClearance

Definition:

```
int GetReverseClearance()
```

This function outputs the reverse clearance in steps.

The function corresponds to serial command `'Zz'`.

SetAnalogueMin

Definition:

```
bool SetAnalogueMin(double analogueMin)
```

This function sets the minimum voltage for the analog input.

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

The function corresponds to serial command `'Q'`.

GetAnalogueMin

Definition:

```
double GetAnalogueMin()
```

This function outputs the minimum voltage for the analog input.

The function corresponds to serial command `'ZQ'`.

SetAngelDeviationMax

Definition:

```
bool SetAngelDeviationMax(int deviation)
```

This function sets the maximum angle deviation between the setpoint position and the encoder value.

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

The function corresponds to serial command `'X'`.

GetAngelDeviationMax

Definition:

```
int GetAngelDeviationMax()
```

This function outputs the maximum angle deviation between the setpoint position and the encoder value.

The function corresponds to serial command `'ZX'`.

SetAnalogueMax

Definition:

```
bool SetAnalogueMax(double analogueMax)
```

This function sets the maximal voltage for the analog input.

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

The function corresponds to serial command `'R'`.

GetAnalogueMax

Definition:

```
double GetAnalogueMax()
```

This function outputs the maximum voltage for the analog input.

The function corresponds to serial command `'ZR'`.

SetPositionType

Definition:

```
bool SetPositionType(int positionType)
```

This function sets the position type.

- 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.

For an exact description of the use, see serial command `'p'`.

GetPositionType

Definition:

```
int GetPositionType(int operationNumber)
```

This function reads out the positioning type.

- 1 corresponds to relative; depends on the operation mode
- 2 corresponds to absolute; depends on the operation mode
- 3 corresponds to an internal reference run;
- 4 corresponds to an external reference run

Here the operationNumber parameter is the record number (travel profile) from which the position type should be read.

For an exact description of the use, see serial command `'p'`.

SetSteps

Definition:

```
bool SetSteps(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 serial command `'s'`.

GetSteps

Definition:

```
int GetSteps(int operationNumber)
```

This function reads out the number of steps.

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

The function corresponds to serial command `'Zs'`.

SetStartFrequency

Definition:

```
bool SetStartFrequency(int startFrequency)
```

This function sets the start frequency.

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

The function corresponds to serial command `'u'`.

GetStartFrequency

Definition:

```
int GetStartFrequency(int operationNumber)
```

This function outputs the start frequency.

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

The function corresponds to serial command `'Zu'`.

SetMaxFrequency2

Definition:

```
bool SetMaxFrequency2(int maxFrequency)
```

This function sets the upper maximum frequency.

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

The function corresponds to serial command `'n'`.

GetMaxFrequency2

Definition:

```
int GetMaxFrequency2(int operationNumber)
```

This function outputs the upper maximum frequency.

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

The function corresponds to serial command `'Zn'`.

SetSuppressResponse

Definition:

```
bool SetSuppressResponse(int suppress)
```

This function activates or deactivates the response suppression on sending.

- suppress = 0: response suppression on
- suppress = 1: response suppression 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 serial command `'|'`.

GetRotationMode

Definition:

```
int GetRotationMode()
```

This function reads the encoder monitoring mode.

- 0 means no monitoring
- 1 means a check at the end
- 2 means a check during a run

The "Check during travel" setting exists for compatibility reasons and is equivalent to the "Check at end" behavior. To actually make a correction during travel, the closed loop mode should be used.

The function corresponds to serial command `'ZU'`.

GetDirection

Definition:

```
int GetDirection(int operationNumber)
```

This function outputs the direction of rotation of the motor.

- 0 corresponds to left
- 1 corresponds to right

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

The function corresponds to serial command `'Zd'`.

SetDirectionReverse

Definition:

```
bool SetDirectionReverse(bool directionReverse)
```

This function sets the reversal in the direction of rotation.

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

The function corresponds to serial command `'t'`.

GetDirectionReverse

Definition:

```
bool GetDirectionReverse(int operationNumber)
```

This function reads out the reversal in the direction of rotation.

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

The function corresponds to serial command `'Zt'`.

SetEncoderDirection

Definition:

```
bool SetEncoderDirection(bool encoderDirection)
```

This function sets the encoder rotation direction. If the encoderDirection parameter is true, the direction of the rotary encoder is 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 serial command `'q'`.

GetEncoderDirection

Definition:

```
bool GetEncoderDirection()
```

This function outputs whether the encoder rotation direction will be reversed.

The function corresponds to serial command `'Zq'`.

GetEncoderRotary

Definition:

```
int GetEncoderRotary()
```

This function reads out the encoder position.

The function corresponds to serial command `'I'`.

SetRampType

Definition:

```
bool SetRampType(int rampType)
```

This function sets the ramp type.

- rampType = 0: trapezoidal ramp
- rampType = 1: sinusoidal ramp
- rampType = 2: jerk-free 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 serial command `':ramp_mode'`.

GetRampType

Definition:

```
int GetRampType()
```

This function outputs the ramp type.

- rampType = 0: trapezoidal ramp
- rampType = 1: sinusoidal ramp
- rampType = 2: jerk-free ramp

The function corresponds to serial command `':ramp_mode'`.

SetJerk

Definition:

```
bool SetJerk(int jerk)
```

This function sets the jerk in 100/s³.

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

The function corresponds to serial command `':b'`.

GetJerk

Definition:

```
int GetJerk(int operationNumber)
```

This function outputs the jerk in 100/s³.

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

The function corresponds to serial command `'Z:b'`.

SetBrakeRamp

Definition:

```
bool SetBrakeRamp(int rampBrake)
```

This function sets the brake 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 serial command `'B'`.

GetBrakeRamp

Definition:

```
int GetBrakeRamp(int operationNumber)
```

This function reads out the brake ramp.

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

The function corresponds to serial command `'ZB'`.

SetBrakeJerk

Definition:

```
bool SetBrakeJerk(int jerk)
```

This function sets the brake jerk in 100/s³.

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

The function corresponds to serial command `':B'`.

GetBrakeJerk

Definition:

```
int GetBrakeJerk(int operationNumber)
```

This function outputs the brake jerk in 100/s³.

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

The function corresponds to serial command `'Z:B'`.

SetQuickStoppRamp

Definition:

```
bool SetQuickStoppRamp(int rampQuickStopp)
```

This function sets the quick stop 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 serial command `'H'`.

GetQuickStoppRamp

Definition:

```
int GetQuickStoppRamp(int operationNumber)
```

This function reads out the quick ramp.

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

The function corresponds to serial command `'ZH'`.

SetRepeat

Definition:

```
bool SetRepeat(int repeats)
```

This function sets the number of repetitions.

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

The function corresponds to serial command `'W'`.

GetRepeat

Definition:

```
int GetRepeat(int operationNumber)
```

This function reads out the number of repetitions.

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

The function corresponds to serial command `'ZW'`.

SetModus8

Definition:

```
bool SetModus8()
```

This function sets the operating mode 14, which corresponds to an internal reference run. In older firmwares, a run in this operating mode was necessary to activate the closed loop mode.

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

SetKalibrierModus

Definition:

```
bool SetKalibrierModus()
```

This function sets the operating mode 17, which performs the calibration run of the CL wizard.

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

SetClosedLoop

Definition:

```
bool SetClosedLoop(int value)
```

This function activates or deactivates the closed loop 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 serial command `':CL_enable'`.

GetClosedLoop

Definition:

```
int GetClosedLoop()
```

This function outputs whether the closed loop mode is activated.

The function corresponds to serial command `':CL_enable'`.

GetCLLoadAngle

Definition:

```
int GetCLLoadAngle(int tripelNumber)
```

This function reads out a load angle of the motor from the closed loop test run.

The tripelNumber parameter is the number (0-9) of the value that should be read out.

The function corresponds to serial command `':CL_la_a'` to `':CL_la_j'`.

GetClosedLoopOlaCurrent

Definition:

```
int GetClosedLoopOlaCurrent(int tripelNumber)
```

This function reads out a correction angle of the current controller from the closed loop test run.

The tripelNumber parameter is the number (0-6) of the value that should be read out.

The function corresponds to serial command `':CL_ola_i_a'` to `':CL_ola_i_g'`.

GetClosedLoopOlaVelocity

Definition:

```
int GetClosedLoopOlaVelocity(int tripelNumber)
```

This function reads out a correction value of the speed controller from the closed loop test run.

The tripelNumber parameter is the number (0-6) of the value that should be read out.

The function corresponds to serial command `':CL_ola_v_a'` to `':CL_ola_v_g'`.

GetClosedLoopOlaLoadAngle

Definition:

```
int GetClosedLoopOlaLoadAngle(int tripelNumber)
```

This function reads out a correction value of the position controller from the closed loop test run.

The tripelNumber parameter is the number (0-6) of the value that should be read out.

The function corresponds to serial command `':CL_ola_l_a'` to `':CL_ola_l_g'`.

SetPositionWindow

Definition:

```
bool SetPositionWindow(int positionWindow)
```

This function sets the tolerance window for the end position in the closed loop 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 serial command `':CL_position_window'`.

GetPositionWindow

Definition:

```
int GetPositionWindow()
```

This function outputs the value for the tolerance window for the end position in the closed loop mode.

The function corresponds to serial command `':CL_position_window'`.

SetPositionWindowTime

Definition:

```
bool SetPositionWindowTime(int time)
```

This function sets the time for the tolerance window of the end position in the closed loop 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 serial command `':CL_position_window_time'`.

GetPositionWindowTime

Definition:

```
int GetPositionWindowTime()
```

This function outputs the value for the time for the tolerance window for the end position in the closed loop mode.

The function corresponds to serial command `':CL_position_window_time'`.

SetFollowingErrorWindow

Definition:

```
bool SetFollowingErrorWindow(int followingErrorWindow)
```

This function sets the maximum allowed following error in the closed loop 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 serial command `':CL_following_error_window'`.

GetFollowingErrorWindow

Definition:

```
int GetFollowingErrorWindow()
```

This function outputs the value for the maximum allowed following error in the closed loop mode.

The function corresponds to serial command `':CL_following_error_window'`.

SetSpeedErrorWindow

Definition:

```
bool SetSpeedErrorWindow(int speedErrorWindow)
```

This function sets the maximum allowed speed deviation in the closed loop 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 serial command `':CL_speed_error_window'`.

GetSpeedErrorWindow

Definition:

```
int GetSpeedErrorWindow()
```

This function outputs the value for the maximum allowed speed deviation in the closed loop mode.

The function corresponds to serial command `':CL_speed_error_window'`.

SetFollowingErrorTimeout

Definition:

```
bool SetFollowingErrorTimeout(int timeout)
```

This function sets the time for the maximum allowed following error in the closed loop 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 serial command `':CL_following_error_timeout'`.

GetFollowingErrorTimeout

Definition:

```
int GetFollowingErrorTimeout()
```

This function outputs the value for the time for the maximum allowed following error in the closed loop mode.

The function corresponds to serial command `':CL_following_error_timeout'`.

SetSpeedErrorTimeout

Definition:

```
bool SetSpeedErrorTimeout(int timeout)
```

This function sets the time for the maximum allowed speed deviation in the closed loop 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 serial command `':CL_speed_error_timeout'`.

GetSpeedErrorTimeout

Definition:

```
int GetSpeedErrorTimeout()
```

This function outputs the value for the time for the maximum allowed speed deviation in the closed loop mode.

The function corresponds to serial command `':CL_speed_error_timeout'`.

SetRotencInc

Definition:

```
bool SetRotencInc(int rotencInc)
```

This function sets the number of encoder increments.

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

The function corresponds to serial command `':CL_rotenc_inc'`.

GetRotencInc

Definition:

```
int GetRotencInc()
```

This function outputs the number of encoder increments.

The function corresponds to serial command `':CL_rotenc_inc'`.

SetBrakeTA

Definition:

```
bool SetBrakeTA(UInt32 brake)
```

This function sets the wait time for switching off the brake 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 serial command `':brake_ta'`.

GetBrakeTA

Definition:

```
int GetBrakeTA()
```

This function outputs the wait time for switching off the brake voltage.

The function corresponds to serial command `':brake_ta'`.

SetBrakeTB

Definition:

```
bool SetBrakeTB(UInt32 brake)
```

This function sets the time in milliseconds between switching off of the brake voltage and enabling of a motor movement.

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

The function corresponds to serial command `':brake_tb'`.

GetBrakeTB

Definition:

```
int GetBrakeTB()
```

This function outputs the between switching off of the brake voltage and enabling of a motor movement.

The function corresponds to serial command `':brake_tb'`.

SetBrakeTC

Definition:

```
bool SetBrakeTC(UInt32 brake)
```

This function sets the wait time for switching off the motor voltage.

The motor current is switched off by resetting the enable input (see Section *2.5.25* "*Setting the function of* the digital inputs").

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

The function corresponds to serial command `':brake_tc'`.

GetBrakeTC

Definition:

```
int GetBrakeTC()
```

This function outputs the wait time for switching off the motor voltage.

The motor current is switched off by resetting the enable input (see Section *2.5.25* "*Setting the function of* the digital inputs").

The function corresponds to serial command `':brake_tc'`.

SetKPsZ

Definition:

```
bool SetKPsZ(int value)
```

This function sets the numerator of the P component of the position 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 serial command `':CL_KP_s_Z'`.

GetKPsZ

Definition:

```
int GetKPsZ()
```

This function outputs the numerator of the P component of the position controller.

The function corresponds to serial command `':CL_KP_s_Z'`.

SetKPsN

Definition:

```
bool SetKPsN(int value)
```

This function sets the denominator of the P component of the position 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 serial command `':CL_KP_s_N'`.

GetKPsN

Definition:

```
int GetKPsN()
```

This function outputs the denominator of the P component of the position controller.

The function corresponds to serial command `':CL_KP_s_N'`.

SetKIsZ

Definition:

```
bool SetKIsZ(int value)
```

This function sets the numerator of the I component of the position 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 serial command `':CL_KI_s_Z'`.

GetKIsZ

Definition:

```
int GetKIsZ()
```

This function outputs the numerator of the I component of the position controller.

The function corresponds to serial command `':CL_KI_s_Z'`.

SetKIsN

Definition:

```
bool SetKIsN(int value)
```

This function sets the denominator of the I component of the position 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 serial command `':CL_KI_s_N'`.

GetKIsN

Definition:

```
int GetKIsN()
```

This function outputs the denominator of the I component of the position controller.

The function corresponds to serial command `':CL_KI_s_N'`.

SetKDsZ

Definition:

```
bool SetKDsZ(int value)
```

This function sets the numerator of the D component of the position 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 serial command `':CL_KD_s_Z'`.

GetKDsZ

Definition:

```
int GetKDsZ()
```

This function outputs the numerator of the D component of the position controller.

The function corresponds to serial command `':CL_KD_s_Z'`.

SetKDsN

Definition:

```
bool SetKDsN(int value)
```

This function sets the denominator of the D component of the position 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 serial command `':CL_KD_s_N'`.

GetKDsN

Definition:

```
int GetKDsN()
```

This function outputs the denominator of the D component of the position controller.

The function corresponds to serial command `':CL_KD_s_N'`.

SetKPvZ

Definition:

```
bool SetKPvZ(int value)
```

This function sets the numerator of the P component of the speed 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 serial command `':CL_KP_v_Z'`.

GetKPvZ

Definition:

```
int GetKPvZ()
```

This function outputs the numerator of the P component of the speed controller.

The function corresponds to serial command `':CL_KP_v_Z'`.

SetKPvN

Definition:

```
bool SetKPvN(int value)
```

This function sets the denominator of the P component of the speed 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 serial command `':CL_KP_v_N'`.

GetKPvN

Definition:

```
int GetKPvN()
```

This function outputs the denominator of the P component of the speed controller.

The function corresponds to serial command `':CL_KP_v_N'`.

SetKIvZ

Definition:

```
bool SetKIvZ(int value)
```

This function sets the numerator of the I component of the speed 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 serial command `':CL_KI_v_Z'`.

GetKIvZ

Definition:

```
int GetKIvZ()
```

This function outputs the numerator of the I component of the speed controller.

The function corresponds to serial command `':CL_KI_v_Z'`.

SetKIvN

Definition:

```
bool SetKIvN(int value)
```

This function sets the denominator of the I component of the speed 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 serial command `':CL_KI_v_N'`.

GetKIvN

Definition:

```
int GetKIvN()
```

This function outputs the denominator of the I component of the speed controller.

The function corresponds to serial command `':CL_KI_v_N'`.

SetKDvZ

Definition:

```
bool SetKDvZ(int value)
```

This function sets the numerator of the D component of the speed 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 serial command `':CL_KD_v_Z'`.

GetKDvZ

Definition:

```
int GetKDvZ()
```

This function outputs the numerator of the D component of the speed controller.

The function corresponds to serial command `':CL_KD_v_Z'`.

SetKDvN

Definition:

```
bool SetKDvN(int value)
```

This function sets the denominator of the D component of the speed 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 serial command `':CL_KD_v_N'`.

GetKDvN

Definition:

```
int GetKDvN()
```

This function outputs the denominator of the D component of the speed controller.

The function corresponds to serial command `':CL_KD_v_N'`.

SetKPcssZ

Definition:

```
bool SetKPcssZ(int value)
```

This function sets the numerator of the P component of the cascading position 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 serial command `':CL_KP_css_Z'`.

GetKPcssZ

Definition:

```
int GetKPcssZ()
```

This function outputs the numerator of the P component of the cascading position controller.

The function corresponds to serial command `':CL_KP_css_Z'`.

SetKPcssN

Definition:

```
bool SetKPcssN(int value)
```

This function sets the denominator of the P component of the cascading position 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 serial command `':CL_KP_css_N'`.

GetKPcssN

Definition:

```
int GetKPcssN()
```

This function outputs the denominator of the P component of the cascading position controller.

The function corresponds to serial command `':CL_KP_css_N'`.

SetKIcssZ

Definition:

```
bool SetKIcssZ(int value)
```

This function sets the numerator of the I component of the cascading position 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 serial command `':CL_KI_css_Z'`.

GetKIcssZ

Definition:

```
int GetKIcssZ()
```

This function outputs the numerator of the I component of the cascading position controller.

The function corresponds to serial command `':CL_KI_css_Z'`.

SetKIcssN

Definition:

```
bool SetKIcssN(int value)
```

This function sets the denominator of the I component of the cascading position 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 serial command `':CL_KI_css_N'`.

GetKIcssN

Definition:

```
int GetKIcssN()
```

This function outputs the denominator of the I component of the cascading position controller.

The function corresponds to serial command `':CL_KI_css_N'`.

SetKDcssZ

Definition:

```
bool SetKDcssZ(int value)
```

This function sets the numerator of the D component of the cascading position 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 serial command `':CL_KD_css_Z'`.

GetKDcssZ

Definition:

```
int GetKDcssZ()
```

This function outputs the numerator of the D component of the cascading position controller.

The function corresponds to serial command `':CL_KD_css_Z'`.

SetKDcssN

Definition:

```
bool SetKDcssN(int value)
```

This function sets the denominator of the D component of the cascading position 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 serial command `':CL_KD_css_N'`.

GetKDcssN

Definition:

```
int GetKDcssN()
```

This function outputs the denominator of the D component of the cascading position controller.

The function corresponds to serial command `':CL_KD_css_N'`.

SetKPcsvZ

Definition:

```
bool SetKPcsvZ(int value)
```

This function sets the numerator of the P component of the cascading speed 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 serial command `':CL_KP_csv_Z'`.

GetKPcsvZ

Definition:

```
int GetKPcsvZ()
```

This function outputs the numerator of the P component of the cascading speed controller.

The function corresponds to serial command `':CL_KP_csv_Z'`.

SetKPcsvN

Definition:

```
bool SetKPcsvN(int value)
```

This function sets the denominator of the P component of the cascading speed 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 serial command `':CL_KP_csv_N'`.

GetKPcsvN

Definition:

```
int GetKPcsvN()
```

This function outputs the denominator of the P component of the cascading speed controller.

The function corresponds to serial command `':CL_KP_csv_N'`.

SetKIcsvZ

Definition:

```
bool SetKIcsvZ(int value)
```

This function sets the numerator of the I component of the cascading speed 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 serial command `':CL_KI_csv_Z'`.

GetKIcsvZ

Definition:

```
int GetKIcsvZ()
```

This function outputs the numerator of the I component of the cascading speed controller.

The function corresponds to serial command `':CL_KI_csv_Z'`.

SetKIcsvN

Definition:

```
bool SetKIcsvN(int value)
```

This function sets the denominator of the I component of the cascading speed 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 serial command `':CL_KI_csv_N'`.

GetKIcsvN

Definition:

```
int GetKIcsvN()
```

This function outputs the denominator of the I component of the cascading speed controller.

The function corresponds to serial command `':CL_KI_csv_N'`.

SetKDcsvZ

Definition:

```
bool SetKDcsvZ(int value)
```

This function sets the numerator of the D component of the cascading speed 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 serial command `':CL_KD_csv_Z'`.

GetKDcsvZ

Definition:

```
int GetKDcsvZ()
```

This function outputs the numerator of the D component of the cascading speed controller.

The function corresponds to serial command `':CL_KD_csv_Z'`.

SetKDcsvN

Definition:

```
bool SetKDcsvN(int value)
```

This function sets the denominator of the D component of the cascading speed 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 serial command `':CL_KD_csv_N'`.

GetKDcsvN

Definition:

```
int GetKDcsvN()
```

This function outputs the denominator of the D component of the cascading speed controller.

The function corresponds to serial command `':CL_KD_csv_N'`.

SetInput1Selection

Definition:

```
bool SetInput1Selection(InputSelection inputSelection)
```

This function sets the function for digital input 1.

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

The function corresponds to serial command `':port_in_a'`.

GetInput1Selection

Definition:

```
InputSelection GetInput1Selection()
```

This function outputs the function for digital input 1.

The function corresponds to serial command `':port_in_a'`.

SetInput2Selection

Definition:

```
bool SetInput2Selection(InputSelection inputSelection)
```

This function sets the function for digital input 2.

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

The function corresponds to serial command `':port_in_b'`.

GetInput2Selection

Definition:

```
InputSelection GetInput2Selection()
```

This function outputs the function for digital input 2.

The function corresponds to serial command `':port_in_b'`.

SetInput3Selection

Definition:

```
bool SetInput3Selection(InputSelection inputSelection)
```

This function sets the function for digital input 3.

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

The function corresponds to serial command `':port_in_c'`.

GetInput3Selection

Definition:

```
InputSelection GetInput3Selection()
```

This function outputs the function for digital input 3.

The function corresponds to serial command `':port_in_c'`.

SetInput4Selection

Definition:

```
bool SetInput4Selection(InputSelection inputSelection)
```

This function sets the function for digital input 4.

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

The function corresponds to serial command `':port_in_d'`.

GetInput4Selection

Definition:

```
InputSelection GetInput4Selection()
```

This function outputs the function for digital input 4.

The function corresponds to serial command `':port_in_d'`.

SetInput5Selection

Definition:

```
bool SetInput5Selection(InputSelection inputSelection)
```

This function sets the function for digital input 5.

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

The function corresponds to serial command `':port_in_e'`.

GetInput5Selection

Definition:

```
InputSelection GetInput5Selection()
```

This function outputs the function for digital input 5.

The function corresponds to serial command `':port_in_e'`.

SetInput6Selection

Definition:

```
bool SetInput6Selection(InputSelection inputSelection)
```

This function sets the function for digital input 6.

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

The function corresponds to serial command `':port_in_f'`.

GetInput6Selection

Definition:

```
InputSelection GetInput6Selection()
```

This function outputs the function for digital input 6.

The function corresponds to serial command `':port_in_f'`.

SetInput7Selection

Definition:

```
bool SetInput7Selection(InputSelection inputSelection)
```

This function sets the function for digital input 7.

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

The function corresponds to serial command `':port_in_g'`.

GetInput7Selection

Definition:

```
InputSelection GetInput7Selection()
```

This function outputs the function for digital input 7.

The function corresponds to serial command `':port_in_g'`.

SetInput8Selection

Definition:

```
bool SetInput8Selection(InputSelection inputSelection)
```

This function sets the function for digital input 8.

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

The function corresponds to serial command `':port_in_h'`.

GetInput8Selection

Definition:

```
InputSelection GetInput8Selection()
```

This function outputs the function for digital input 8.

The function corresponds to serial command `':port_in_h'`.

SetOutput1Selection

Definition:

```
bool SetOutput1Selection(OutputSelection outputSelection)
```

This function sets the function for digital output 1.

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

The function corresponds to serial command `':port_out_a'`.

GetOutput1Selection

Definition:

```
OutputSelection GetOutput1Selection()
```

This function outputs the function for digital output 1.

The function corresponds to serial command `':port_out_a'`.

SetOutput2Selection

Definition:

```
bool SetOutput2Selection(OutputSelection outputSelection)
```

This function sets the function for digital output 2.

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

The function corresponds to serial command `':port_out_b'`.

GetOutput2Selection

Definition:

```
OutputSelection GetOutput2Selection()
```

This function outputs the function for digital output 2.

The function corresponds to serial command `':port_out_b'`.

SetOutput3Selection

Definition:

```
bool SetOutput3Selection(OutputSelection outputSelection)
```

This function sets the function for digital output 3.

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

The function corresponds to serial command `':port_out_c'`.

GetOutput3Selection

Definition:

```
OutputSelection GetOutput3Selection()
```

This function outputs the function for digital output 3.

The function corresponds to serial command `':port_out_c'`.

SetOutput4Selection

Definition:

```
bool SetOutput4Selection(OutputSelection outputSelection)
```

This function sets the function for digital output 4.

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

The function corresponds to serial command `':port_out_d'`.

GetOutput4Selection

Definition:

```
OutputSelection GetOutput4Selection()
```

This function outputs the function for digital output 4.

The function corresponds to serial command `':port_out_d'`.

SetOutput5Selection

Definition:

```
bool SetOutput5Selection(OutputSelection outputSelection)
```

This function sets the function for digital output 5.

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

The function corresponds to serial command `':port_out_e'`.

GetOutput5Selection

Definition:

```
OutputSelection GetOutput5Selection()
```

This function outputs the function for digital output 5.

The function corresponds to serial command `':port_out_e'`.

SetOutput6Selection

Definition:

```
bool SetOutput6Selection(OutputSelection outputSelection)
```

This function sets the function for digital output 6.

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

The function corresponds to serial command `':port_out_f'`.

GetOutput6Selection

Definition:

```
OutputSelection GetOutput6Selection()
```

This function outputs the function for digital output 6.

The function corresponds to serial command `':port_out_f'`.

SetOutput7Selection

Definition:

```
bool SetOutput7Selection(OutputSelection outputSelection)
```

This function sets the function for digital output 7.

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

The function corresponds to serial command `':port_out_g'`.

GetOutput7Selection

Definition:

```
OutputSelection GetOutput7Selection()
```

This function outputs the function for digital output 7.

The function corresponds to serial command `':port_out_g'`.

SetOutput8Selection

Definition:

```
bool SetOutput8Selection(OutputSelection outputSelection)
```

This function sets the function for digital output 8.

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

The function corresponds to serial command `':port_out_h'`.

GetOutput8Selection

Definition:

```
OutputSelection GetOutput8Selection()
```

This function outputs the function for digital output 8.

The function corresponds to serial command `':port_out_h'`.

SetFeedConstNum

Definition:

```
bool SetFeedConstNum(int feedConstNum)
```

This function sets the numerator of the feed rate.

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

The function corresponds to serial command `':feed_const_num'`.

GetFeedConstNum

Definition:

```
int GetFeedConstNum()
```

This function outputs the numerator of the feed rate.

The function corresponds to serial command `':feed_const_num'`.

SetFeedConstDenum

Definition:

```
bool SetFeedConstDenum(int feedConstDenum)
```

This function sets the denominator of the feed rate.

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

The function corresponds to serial command `':feed_const_denum'`.

GetFeedConstDenum

Definition:

```
int GetFeedConstDenum()
```

This function outputs the denominator of the feed rate.

The function corresponds to serial command `':feed_const_denum'`.

SetCurrentPeak

Definition:

```
bool SetCurrentPeak(int currentPeak)
```

This function sets the current peak value for BLDC.

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

The function corresponds to serial command `':ipeak'`.

GetCurrentPeak

Definition:

```
int GetCurrentPeak()
```

This function outputs the current peak value for BLDC.

The function corresponds to serial command `':ipeak'`.

SetCurrentTime

Definition:

```
bool SetCurrentTime(int currentTime)
```

This function sets the current time constant for BLDC.

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

The function corresponds to serial command `':itime'`.

GetCurrentTime

Definition:

```
int GetCurrentTime()
```

This function outputs the current time constant for BLDC.

The function corresponds to serial command `':itime'`.

4.4 Programming examples

Introduction

Some examples for the use of the commandsPD41 function library are provided in the NanoPro installation directory in the SDK/example subdirectory. All examples are implemented as projects for Microsoft Visual Studio. All examples demonstrate the interaction with 2 controllers at different serial interfaces. A short list of the examples follows.

CsharpExample

This example is implemented in the C# programming language and realized as a Visual Studio 2005 project.

ManagedC++Example:

This example is implemented in the C++ programming language using Managed Code and is realized as a Visual Studio 2008 project.

UnmanagedC++Example:

This example is implemented in the C++ programming language using Unmanaged Code and is realized as a Visual Studio 2008 project. Unlike the other examples, this example does not have a graphical user interface.

VBExample:

This example is implemented in the Visual Basic programming language and realized as a Visual Studio 2005 project.

Index

S

T