Schunk Motion Protocol for Python 3

Release

Matthias Geier

June 02, 2014

Contents

Documentation: http://schunk.rtfd.org/

Code: http://github.com/spatialaudio/schunk/

Schunk Motion Protocol manual: http://www.schunk.com/schunk_files/attachments/MotionControl_en_2010-03.pdf

1 Disclaimer

This is *not* a commercial product and the author has *no relation whatsoever* to SCHUNK GmbH & Co. KG[1].

Use at your own risk!

2 Devices

Only 1 device was tested: Schunk PR-70 Servo Electric Swivel Unit[2].

Defaults for this device: RS232, baudrate=9600, module ID 11 (0x0B).

[1] http://schunk.com/
[2] http://tinyurl.com/schunk-pr/

Other devices may or may not work.

3 Requirements

Python[3] version 3.x is required.

Typically, PySerial[4] handles the serial connection, but any library with a similar API can be used.

py.test[5] is used for the tests.

4 Limitations

Only a subset of the Schunk Motion Protocol is supported.

Only the direct response to a command can be obtained, *impulse messages* are not supported. One exception is the "CMD POS REACHED" impulse message which is used to realize movement commands which are waiting until the movement is finished, e.g. `move_pos_blocking()`.

Only floating point unit systems are supported.

The connection is opened and closed for each message. Keeping the connection open is not supported.

Only the RS232 protocol is implemented.

5 Installation

```
python3 setup.py install
```

Alternatively, you can just copy `schunk.py` to your working directory.

If you want to make changes to `schunk.py`, you should type:

```
python3 setup.py develop
```

or, alternatively:

```
pip3 install -e .
```

6 Tests

Tests are implemented using py.test[6], run this in the main directory:

```
python3 setup.py test
```

7 Examples

This should get you started:

[3] http://www.python.org/
[4] http://pyserial.sf.net/
[5] http://pytest.org/
[6] http://pytest.org/

```
import schunk
import serial

mod = schunk.Module(schunk.RS232Connection(
    0x0B, serial.Serial, port=0, baudrate=9600, timeout=1))

mod.move_pos(42)
```

Use the ID of your Schunk module instead of `0x0B`.

See the documentation of PySerial[7] for all possible serial port options. You probably only have to change `port`, e.g. `port='/dev/ttyS1'` or `port='COM3'`.

It is useful to specify a *timeout*, otherwise you may have to wait forever for the functions to return if there is an error. On the other hand, if you want to use the blocking commands (`*_blocking()`), you should disable the timeout (or make it longer than the expected movement times).

If the parameters for your setup don't change, you can write them into a separate file, e.g. with the name `myschunk.py`:

```
import schunk
import serial

module1 = schunk.Module(schunk.RS232Connection(
    0x0B, serial.Serial, port=0, baudrate=9600, timeout=1))
```

and then use it like this in all our scripts:

```
from myschunk import module1
module1.move_pos(42)
```

The file `myschunk.py` must be in the current directory for this to work.

If you are an object oriented kind of person, you can of course also write your own class:

```
import schunk
import serial

class MySchunkModule(schunk.Module):
    def __init__(self):
        super().__init__(schunk.RS232Connection(
            0x0B, serial.Serial, port=0, baudrate=9600, timeout=1))

module1 = MySchunkModule()
module1.move_pos(42)
```

8 API Documentation

Schunk Motion Protocol for Python 3.

Documentation: http://schunk.rtfd.org/

Code: http://github.com/spatialaudio/schunk/

Schunk Motion Protocol manual: http://www.schunk.com/schunk_files/attachments/MotionControl_en_2010-03.pdf

[7] http://pyserial.sf.net/

Example

```
import schunk
import serial

mod = schunk.Module(schunk.RS232Connection(
    0x0B, serial.Serial, port=0, baudrate=9600, timeout=1))

mod.move_pos(42)
```

class schunk.**Module**(*connection*)

Create an object for controlling a Schunk module.

Parameters **connection** – Something that has an `open()` method which returns a coroutine. This coroutine must accept a bytes object and send it to a Schunk module, read the response (taking D-Len into account) and yield the response (and further messages) as a bytes object.

`RS232Connection` happens to do exactly that.

reference()

2.1.1 CMD REFERENCE (0x92).

A reference movement is completed.

move_pos(*position*, *velocity=None*, *acceleration=None*, *current=None*, *jerk=None*)

2.1.3 MOVE POS (0xB0).

Parameters

- **position** (*float*) – Absolute position.
- **velocity, acceleration, current, jerk** (*float, optional*) – If one of them is not specified, all following arguments must not be specified either.

Returns *float* – Estimated time to reach *position*. If the time cannot be estimated, 0.0 is returned.

See also:

`move_pos_blocking()`, `move_pos_rel()`, `set_target_vel()`, `set_target_acc()`, `set_target_cur()`, `set_target_jerk()`

move_pos_blocking(*position*, *velocity=None*, *acceleration=None*, *current=None*, *jerk=None*)

Move to position and wait until position is reached.

Note: *Impulse messages* must be activated for this to work, see `toggle_impulse_message()` and `communication_mode`.

This applies to all `*_blocking()` methods.

Returns *float* – The final position.

See also:

`move_pos()`

move_pos_rel(*position*, *velocity=None*, *acceleration=None*, *current=None*, *jerk=None*)

2.1.4 MOVE POS REL (0xB8).

Parameters

- **position** (*float*) – Relative position.
- **velocity, acceleration, current, jerk** (*float, optional*) – If one of them is not specified, the following must not be specified either.

Returns *float* – Estimated time to reach *position*. If the time cannot be estimated, 0.0 is returned.

See also:

`move_pos_rel_blocking()`, `move_pos()`, `set_target_vel()`, `set_target_acc()`, `set_target_cur()`, `set_target_jerk()`

move_pos_rel_blocking(*position*, *velocity=None*, *acceleration=None*, *current=None*, *jerk=None*)

Move to relative position and wait until position is reached.

Returns *float* – The actual relative motion.

See also:

`move_pos_rel()`

move_pos_time(*position*, *velocity=None*, *acceleration=None*, *current=None*, *time=None*)

2.1.5 MOVE POS TIME (0xB1).

See also:

`move_pos_time_blocking()`, `move_pos()`, `set_target_time()`

move_pos_time_blocking(*position*, *velocity=None*, *acceleration=None*, *current=None*, *time=None*)

Move to position and wait until position is reached.

Returns *float* – The final position.

See also:

`move_pos_time()`

move_pos_time_rel(*position*, *velocity=None*, *acceleration=None*, *current=None*, *time=None*)

2.1.6 MOVE POS TIME REL (0xB9).

See also:

`move_pos_time_rel_blocking()`, `move_pos_rel()`, `set_target_time()`

move_pos_time_rel_blocking(*position*, *velocity=None*, *acceleration=None*, *current=None*, *time=None*)

Move to position and wait until position is reached.

Returns *float* – The actual relative motion.

See also:

`move_pos_time_rel()`

set_target_vel(*velocity*)

2.1.14 SET TARGET VEL (0xA0).

Initially, the target velocity is set to 10% of the maximum.

set_target_acc(*acceleration*)

2.1.15 SET TARGET ACC (0xA1).

Initially, the target acceleration is set to 10% of the maximum.

set_target_jerk(*jerk*)

2.1.16 SET TARGET JERK (0xA2).

Initially, the target jerk is set to 50% of the maximum.

set_target_cur(*current*)

2.1.17 SET TARGET CUR (0xA3).

Initially, the target current is set to the nominal current.

set_target_time(*time*)
2.1.18 SET TARGET TIME (0xA4).

stop()
2.1.19 CMD STOP (0x91).

toggle_impulse_message()
2.2.6 CMD TOGGLE IMPULSE MESSAGE (0xE7).

Note: *Impulse messages* must be switched on for `*_blocking()`, e.g. `move_pos_blocking()`.

Returns *bool* – `True` if impulse messages were switched on, `False` if they were switched off.

config
2.3.1 SET CONFIG (0x81) / 2.3.2 GET CONFIG (0x80).

The *config* object has several attributes which can be queried and changed. Except where otherwise noted, the new settings are immediately stored in the EEPROM but are only applied after the module has been restarted.

Some options are read-only, some can only be set as "Profi" user. See `change_user()`.

module_type bytes

firmware_version int

protocol_version int

hardware_version int

firmware_date bytes

eeprom bytes
All configuration data is read/written in one process. Depending on the type of user certain data might not be written. After successful writing of the data, the module is rebooted.

Note: This command should not be used with one's own applications, as the structure of the data to be received/sent is not known.

---

module_id int (1..255)

group_id int (1..255)

rs232_baudrate int (1200, 2400, 4800, 9600, 19200, 38400)

can_baudrate int (50, 100, 125, 250, 500, 800, 1000)

communication_mode int
See `communication_modes`.

unit_system int
See `unit_systems`.

soft_high float
The transferred value is not written to the EEPROM. The settings are applied immediately.

soft_low float
The transferred value is not written to the EEPROM. The settings are applied immediately.

gear_ratio float
The Gear Ratio 1 is changed (the command has no use with an integer unit system). The transferred value is written to the EEPROM and applied immediately.

max_velocity float

max_acceleration **float**

max_current **float**

nom_current **float**

max_jerk **float**

offset_phase_a **int**

offset_phase_b **int**

data_crc **int**
A CRC16 over all variable and not module specified paramenters (like serial number, current offset).

reference_offset **float**

serial_number **int**

order_number **int**

get_state()
2.5.1 GET STATE (0x95).

Return the module status and other information.

The time parameter (to get state repeatedly) is disabled (because impulse messages are not supported). The mode parameter is always set to request everything (position, velocity and current).

Returns

- **position, velocity, current** (*float*) – Dito.
- **status** (*dict*) – See `decode_status()`.
- **error_code** (*int*) – See `error_codes` for a mapping to strings.

reboot()
2.5.2 CMD REBOOT (0xE0).

change_user(*password=None*)
2.5.6 CHANGE USER (0xE3).

If no password is specified - or if the password is wrong - the user is changed to "User". The default password for "Profi" is "Schunk", but don't tell anyone!

After a reboot, the default user is "User".

check_mc_pc_communication()
2.5.7 CHECK MC PC COMMUNICATION (0xE4).

Returns *bool* – `True` on success.

check_pc_mc_communication()
2.5.8 CHECK PC MC COMMUNICATION (0xE5).

Returns *bool* – `True` on success.

ack()
2.8.1.4 CMD ACK (0x8B).

Acknowledgement of a pending error message.

get_detailed_error_info()
2.8.1.5 GET DETAILED ERROR INFO (0x96).

Returns

- **command** (*{"ERROR", "WARNING", "INFO"}*)
- **error_code** (*int*) – See `error_codes` for a mapping to strings.
- **data** (*float*) – The value can be interpreted by the Schunk Service.

Raises `SchunkError` – If no error is active, or no detailed information is available, the command is raising an exception saying: `INFO FAILED (0x05)`.

exception `schunk.`**`SchunkError`**
This exception is raised on all kinds of errors.

`schunk.`**`coroutine`**(*func*)
Decorator for generator functions that calls next() initially.

class `schunk.`**`RS232Connection`**(*id*, *serialmanager*, **args*, ***kwargs*)
Prepare a serial connection using the RS232 protocol.

This can be used to initialize a `Module`.

The connection is opened with `open()`.

Parameters

- **id** (*int*) – Module ID of the Schunk device.
- **serialmanager** – A callable (to be called with `*args` and `**kwargs`) that must return a context manager which in turn must have `read()` and `write()` methods (and it should close the connection automatically in the end).

  This is typically `serial.Serial` from PySerial[8], but anything with a similar API can be used.

  Note: there should be a timeout, otherwise you may have to wait forever for the functions to return if there is an error. On the other hand, receiving multiple responses only works if there is no timeout in between. Multiple responses are needed for the blocking movement commands, e.g. `Module.move_pos_blocking()`.

- ***args, **kwargs** – All further arguments are forwarded to *serialmanager*.

See also:

`Module`

Examples

Using PySerial[9]:

```
>>> import serial
>>> conn = RS232Connection(0x0B, serial.Serial, port=0,
...                        baudrate=9600, timeout=1)
```

`open`()
Open an RS232 connection.

A coroutine (a.k.a. generator object) is returned which can be used to send and receive one or more data frames.

Calling `.send(data)` on this coroutine creates an RS232 frame around *data*, sends it to the module and waits for a response.

data must have at least two bytes, D-Len and command code. The (optional) rest are parameters. 2 Group/ID bytes are added in the beginning and 2 CRC bytes in the end. The first byte is always 0x05 (= message from master to module), the second byte holds the module ID.

When receiving a response, the 2 CRC bytes are checked (and removed), as well as the 2 Group/ID bytes.

[8] http://pyserial.sf.net/
[9] http://pyserial.sf.net/

The connection is kept open and the coroutine can be invoked repeatedly to receive further data frames. Use `.send(None)` or the built-in `next()` function to receive a data frame without sending anything.

When the desired number of frames has been received, the connection has to be closed with the generator's `close()` method.

Yields *bytes* – Response data received from the module, including D-Len and command code.

If the first RS232 byte indicates an error (0x03), the response is returned normally and the error has to be handled in the calling function. Error responses always have a D-Len of 2, i.e. they have 3 bytes: D-Len, command code and error code.

See also:

`crc16()`

exception `schunk.`**`SchunkRS232Error`**

Exception class for errors related to RS232 connections.

It is derived from `SchunkError`, so it is normally sufficient to check only for this:

```
try:
    ...
    # Something that may throw SchunkError or SchunkRS232Error
    ...
except SchunkError as e:
    # Do something with e
    ...
```

`schunk.`**`decode_status`**(*status*)

This is internally used in `Module.get_state()`.

```
>>> status = decode_status(0x03)
>>> from pprint import pprint
>>> pprint(status)  # to get pretty dict display
{'brake': False,
 'error': False,
 'move_end': False,
 'moving': True,
 'position_reached': False,
 'program_mode': False,
 'referenced': True,
 'warning': False}
```

`schunk.`**`crc16_increment`**(*crc*, *data*)

Incrementally calculate CRC16.

Implementation according to Schunk Motion Protocol documentation.

Parameters

- **crc** (*int*) – Previous CRC16 (2 bytes)
- **data** (*int*) – Data to append (1 byte)

Returns *int* – New CRC16 (again 2 bytes) after appending *data*.

See also:

`crc16()`

`schunk.`**`crc16`**(*data*)

Calculate CRC16 for a sequence of bytes.

Parameters **data** (*iterable of integers (0..255) or bytes*) – A sequece of bytes.

Returns *bytes* – CRC16 of *data* (2 bytes, little endian, a.k.a. '<H').

See also:

`crc16_increment()`

schunk.**communication_modes** = {0: 'AUTO', 1: 'RS232', 2: 'CAN', 3: 'Profibus DPV0', 4: 'RS232 Silent'}
Available communication modes.

See `Module.config`.

schunk.**unit_systems** = {0: '[mm]', 1: '[m]', 2: '[Inch]', 3: '[rad]', 4: '[Degree]', 5: '[Intern]', 6: '[µm] Integer', 7: '[µ
Available unit systems.

See `Module.config`.

Note: This Python module doesn't support integer unit systems.

---

schunk.**error_codes** = {0: 'NO ERROR', 1: 'INFO BOOT', 2: 'INFO NO FREE SPACE', 3: 'INFO NO RIGHTS', 4:
Error codes.

See also `Module.get_state()`.

Note: Error in Schunk manual: key 0xE4 (= 228) is not unique!
