README

Perry Kundert

January 2, 2015

Contents

1 Control SMC Electric Actuators from Python **1**
- 1.1 SMC Actuator and Gateway Protocol . . . 2
- 1.2 Communication Limits and Hardware . . . 2
- 1.3 Installing . . . 3
- 1.4 Positioning . . . 3
  - 1.4.1 `smc.smc_modbus` . . . 3
  - 1.4.2 `.position` – Complete operation, Initiate new position 4
  - 1.4.3 `.complete` – Check for completion . . . 5
  - 1.4.4 `.outputs` – Set/clear outputs (Coils) . . . 6
  - 1.4.5 `.status` – Return full status and position data . . . 6
  - 1.4.6 Command- or Pipe-line usage . . . 8
- 1.5 SMC Gateway Simulator . . . 9

1 Control SMC Electric Actuators from Python

SMC produces a wide variety of Electric Actuators. These are controllable via Gateway Units, providing access via several industrial protocols such as DeviceNet, EtherNet/IP and ProfiBus.

These protocols are typically accessed using industrial control software and devices such as PLCs.

To control your SMC actuators directly from a Python program, you can skip the SMC Gateway, and directly access the actuator Controller using its native protocol. You can reduce the expense of your installation by eliminating the SMC Gateway, and you can run your own Python software on

the same industrial computer used to communicate with the SMC actuator Controllers.

1.1 SMC Actuator and Gateway Protocol

The underlying protocol spoken by the SMC Electric Actuator Controllers themselves (and also the associated SMC Gateways) is simply Modbus/RTU, over an RS-485 serial multi-drop network.

Using `cpppo_positioner`, you can directly access multiple SMC electric actuator controllers (**without** an SMC Gateway), and:

- issue multiple positioning operations in progress simultaneously
- monitor any positioning operation for completion
- set and clear any actuator Controller outputs
- monitor all Controller status flags

The `cpppo_positioner` module allows control of the position of a set of actuators by initiating a connection to the RS-485 communication channel and issuing new position directives via each actuator's controller. The current state is continuously polled via Modbus/RTU reads, and data updates and state changes are performed via Modbus/RTU writes.

1.2 Communication Limits and Hardware

The recommended hardware platform is the Lanner LEC-3013 industrial solid-state PC, which can be configured with up to 8 RS-485 ports, and communicate with up to 12 actuators per port (to minimize polling latency). In addition, the SMC LEC-W2 "Controller setting kit" comes with a USB-RS485 cable which may be used to communicate with additional actuators.

A custom harness is available with the the custom SMC RJ45 plug to RS-485 serial wiring, and an Emergency Stop button. One is required for each separate RS-485 connection (up to 12 actuators).

Therefore, as many as 100 SMC actuators could be controlled by a single `cpppo_positioner` installation running on a Lanner LEC-3013. If low latency (time to detect status changes) is not required, controlling even more than 100 actuators may be possible.

1.3 Installing

Cpppo$_{\text{positioner}}$ requires `pymodbus` version 1.3.0; this requires you to install it from source:

```
$ git clone git@github.com:bashwork/pymodbus.git
$ cd pymodbus
$ python setup.py install
```

Clone the `cpppo_positioner.git` repository, and run the setup.py installer:

```
$ git clone git@github.com:pjkundert/cpppo_positioner.git
$ cd cpppo_positioner
$ python setup.py install
$ python
Python 2.7.6 (default, Sep  9 2014, 15:04:36)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.39)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> from cpppo_positioner import smc
>>>
```

1.4 Positioning

A Python API is provided to implement positioning control for SMC actuators.

1.4.1 `smc.smc_modbus`

This class is the gateway for accessing multiple SMC positioning actuators connected via RS-485 serial. The serial port parameters are `/dev/ttyS1`, 38400 Baud, 8 bits, 1 stop, no parity, and a .25s poll rate. These can all be specified as keyword arguments. See `cpppo_positioner/smc.py` for details.

```
from cpppo_positioner import smc
gateway           = smc.smc_modbus()
```

keyword	description
address	The serial port device address, default /dev/ttyS1
timeout	The RS-485 I/O timeout, default .1s
baudrate	Default 38,400
stopbits	Default 1
bytesize	Default 8
parity	Default is no parity
rate	Adjust to optimize load, RS-485 capacity, latency, default .25s

Nothing will be polled until the first attempt to interact with an actuator. Once an actuator is identified, the `smc_modbus` class will attempt to poll it at the specified `rate`

If an operation raises an Exception, it is expected that you will discard the instance and create a new one.

1.4.2 `.position` – Complete operation, Initiate new position

The .position method checks that any current position operation is complete, and then sends any new position data, starting the new position operation. If no new data is provided (eg. only `actuator` and/or `timeout` provided), then only the operation completion is checked; no new positioning operation is initiated.

```
gateway.position( actuator=1, timeout=10.0, position=12345, speed=100, ... )
```

keyword	description
actuator	The actuator number to operate on
timeout	Allowed number of seconds to complete (forever if None)
svoff	If positioning complete, turn off servo
noop	Don't return home, write new step data but don't initiate

The full set of positioning parameters defined by the SMC actuator is:

keyword	units	description
movement_mode		1: absolute, 2: relative
speed	mm/s	1-65535
position	.01 mm	+/-2147483647
acceleration	mm/s^2	1-65535
deceleration	mm/s^2	1-65535
pushing_force	%	0-100
trigger_level	%	0-100
pushing_speed	mm/s	1-65535
moving_force	%	0-300
area_1	.01 mm	+/-2147483647
area_2	.01 mm	+/-2147483647
in_position	.01 mm	1-2147483647

It is recommended to specify all the values at least for the initial positioning; any values not specified in subsequent position calls will not be changed.

To just confirm that a previous positioning operation has completed:

```
.position( actuator=1, timeout=3 ) # success if completes w/in 3 seconds
.position( actuator=1, svoff=True, timeout=3 ) # ... and turn off servo
```

To check for completion and then return to home position within timeout:

```
.position( actuator=1, home=True, timeout=3 )
```

To check for completion then (without returning to home position), initiate new positioning operation to 150.00mm, within timeout of 3 seconds:

```
.position( actuator=1, position=15000, timeout=3 )
```

1.4.3 `.complete` – Check for completion

Confirms that any previous actuator positioning operation is complete, by monitoring the BUSY flag (not the INP flag, as erroneously indicated by the LEC Modbus RTU op Manual.pdf documentation).

If you wish, you may invoke the `.complete` method directly (instead of implicitly at the beginning of every `.position` invocation).

keyword	description
actuator	The actuator number to operate on
timeout	Allowed number of seconds to complete (forever if None)
svoff	If positioning complete, turn off servo

To check for completion and then disable servo within timeout of 3 seconds:

```
complete( actuator=1, svoff=True, timeout=3 )
```

1.4.4 `.outputs` – Set/clear outputs (Coils)

Modifies one or more named outputs (Coils) on the specified actuator. An integer actuator number is required, followed by optional flags (a variable number of positional parameters)

flags	description
IN[0-5]	
HOLD	
SVON	
DRIVE	
RESET	
SETUP	
JOG_MINUS	
JOG_PLUS	
INPUT_INVALID	

1.4.5 `.status` – Return full status and position data

Returns the current complete set of status and data values for the actuator. If any value has not yet been polled, it will be `None`.

keyword	description
actuator	The actuator number to operate on

Here is an example (formatted for readability):

```
.status( actuator=1 )
{
    "X40_OUT0": false,
    "X41_OUT1": false,
    "X42_OUT2": false,
    "X43_OUT3": false,
    "X44_OUT4": false,
    "X45_OUT5": false,
    "X48_BUSY": false,
    "X49_SVRE": false,
```

```
    "X4A_SETON": false,
    "X4B_INP": false,
    "X4C_AREA": false,
    "X4D_WAREA": false,
    "X4E_ESTOP": false,
    "X4F_ALARM": false,
    "Y10_IN0": false,
    "Y11_IN1": false,
    "Y12_IN2": false,
    "Y13_IN3": false,
    "Y14_IN4": false,
    "Y15_IN5": false,
    "Y18_HOLD": false,
    "Y19_SVON": false,
    "Y1A_DRIVE": false,
    "Y1B_RESET": false,
    "Y1C_SETUP": false,
    "Y1D_JOG_MINUS": false,
    "Y1E_JOG_PLUS": false,
    "Y30_INPUT_INVALID": false,
    "acceleration": 0,
    "area_1": 0,
    "area_2": 0,
    "current_position": 0,
    "current_speed": 0,
    "current_thrust": 0,
    "deceleration": 0,
    "driving_data_no": 0,
    "in_position": 0,
    "movement_mode": 0,
    "moving_force": 0,
    "operation_start": 0,
    "position": 0,
    "pushing_force": 0,
    "pushing_speed": 0,
    "speed": 0,
    "target_position": 0,
    "trigger_level": 0
}
```

1.4.6 Command- or Pipe-line usage

An executable module entry point (`python -m cpppo_positioner`), and a convenience executable script (`cpppo_positioner`) are supplied.

If your application generates a stream of actuator position data, or if you have some manual positions you wish to move to, you can use the command-line interface. You may supply one or more actuator positions in blobs of JSON data (an actual position would have more entries, such as `acceleration`, `deceleration`, `timeout`, ...):

```
$ position='{ "actuator": 0, "position": 12345, "speed": 100 }'
```

These positions may be supplied either as single parameters on the command line, or as separate lines of input (if standard input is selected, by supplying a '-' option):

```
$ python -m cpppo_positioner --address gateway -v "$position"
$ echo "$position" | cpppo_positioner -v -
```

JSON type	description
number	delay for the specified seconds
list	set/clear the named outputs [<actuator>, "FLAG", "flag"]
dict	actuate the position (just check for completion if no position)

Here is an example of setting then clearing the RESET output, then beginning a position operation, and then waiting for it to complete in 10 seconds:

```
$ python -m cpppo_positioner -vv '[1,"RESET"]' 1 '[1,"reset"]' 1 \
   '{"actuator":1, "position":1000, ...}' '{"actuator":1,"timeout":10}'
```

See `cpppo_positioner/main.example` for the text of such an example (run it using `bash main.example`, if you want to try it – it operates actuator #1!)

- Quoting double-quotes on Windows Powershell
  Note that on Windows Cmd or Powershell, it is very difficult to quote double-quote characters in strings. In Powershell, you need to use the bash-slash + back-tick before each double-quote. Unexpectedly, using a single-quoted string does **not** allow you to contain double-quotes.

  You can get double quotes into a string:

```
PS > $position = '{ "actuator": 0, "position": 12345, "speed": 100 }'
PS > $position
'{ "actuator": 0, "position": 12345, "speed": 100 }'
PS >
```

However, when you try to use them, they are re-interpreted on inclusion in a command:

```
PS > python -m cpppo_positioner -v "$position"
... Invalid position data: { actuator: 0, position: 12345, speed: 100 };
    Expecting property name: line 1 column 3 (char 2)
```

So, the only way to do this is to use the strange back-slash + back-tick double-escape, directly as a command-line argument:

```
PS > python -m cpppo_positioner -v '{ \`"actuator\`": 0, ... }'
```

Recommendation: use Linux or Mac, or install Cygwin and use bash on Windows. Trust me; this is just the tip of the iceberg...

1.5 SMC Gateway Simulator

A basic simulator of some of the Modbus/RTU I/O behaviour of an SMC actuator is implemented for testing purposes. To use, disconnect the SMC actuators, and re-connect the Lanner's loop-back plug to the RS-485 harness RJ45 socket.

Ensure that either you have installed the $\text{cpppo}_{\text{positioner}}$, **or** are in the directory containing the cloned $\text{cpppo}_{\text{positioner}}$ repository): To simulate an SMC positioning actuator 1 on /dev/ttyS0:

```
$ python -m cpppo_positioner.simulator -v /dev/ttyS0 1
```