cpppo_positioner: SMC Actuator Control via RS-485

Perry Kundert

2015-03-03 13:44:44

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 Python and `cpppo-positioner`

Install a Python 3.9+ installation of your choice to run code using `cpppo_positioner`. On Windows, start a Terminal with PowerShell as Administrator, and run:

```
winget install -e --id Python.Python3.12 --scope machine
```

1.3.1 From PyPI

It is recommended that you don't install this (or *any*) Python packages globally; create a Python `venv` virtual environment:

```
$ python -m venv SMC-Project
$ . ./SMC-Project/bin/activate
$ (SMC-Project) $ python -m pip install cpppo-positioner
```

On Windows Powershell:

```
PS> python -m venv SMC-Project
PS> Set-ExecutionPolicy -ExecutionPolicy RemoteSigned -Scope CurrentUser
PS> ./SMC-Project/Scripts/Activate.ps1
(SMC-Project) PS> python -m pip install cpppo_positioner
```

Note that the PyPI module name for the `cpppo_positioner` module is: `cpppo-positioner` (a dash, not an underscore). This is the PyPI convention for any Python module containing underscores.

Now, every time you wish to use your Python 3 + `cpppo_positioner`, activate the `venv` before attempting to run your Python code that attempts to import and use `cpppo_positioner`.

1.3.2 From Source

Cpppo_positioner requires `pymodbus` version 3.8.1. Until `pymodbus` integrates some fixes, it will **not** support RS-485 multi-drop; polling or writing multiple RS-485 servers (devices) from a single client will not work! Install `https://github.com/pjkundert/pymodbus/tree/fix/decode` if RS-485 multi-drop support is required.

Clone the `cpppo_positioner.git` repository:

```
$ git clone git@github.com:pjkundert/cpppo_positioner.git
$ cd cpppo_positioner
```

Obtain a Python 3.9+ interpreter, and create and enable a Python virtual environment (uses `venv`) containing the `cpppo_positioner` module:

```
make venv
```

Alternatively, set up your own Python 3 installation that allows `pip` installs, and:

```
$ python -m pip install .[all,tests]
$ python
>>> from cpppo_positioner import smc_modbus
>>>
```

1.3.3 Testing

Once you have a working Python 3 venv and activated it, and installed `cpppo-positioner` in it, you can use your cloned repository and some simulated RS-485 Pseudo-TTYs to verify that it passes its unit tests, verifying that the basic API is operational.

1. Create a `venv` (eg. `SMC-Project`) and activate it with `. ./SMC-Project/bin/activate`
2. Install `cpppo-positioner` in it with `python -m pip install cpppo-positioner`
3. Clone the `https://github.com/pjkundert/cpppo_positioner.git` repository to eg. `~/src/...`
4. Start some PTYs eg. `ttyV0...` in a terminal using `python -m cpppo_positioner.ttyV-setup`
5. Run the unit tests in the same directory where the `ttyV0...` files are using the repo:

```
(SMC-Project) $ SERIAL_TEST=ttyV python -m pytest -v --capture=no --log-cli-level=INFO \
    -k smc_ ~/src/cpppo_positioner
```

This will run the `smc_` unit tests. If you skip the `--log-cli-level=INFO`, you'll see something like:

```
=============================== test session starts ===============================
platform darwin -- Python 3.12.3, pytest-8.3.4, pluggy-1.5.0 -- /private/tmp/SMC-Project/bin/python3
cachedir: .pytest_cache
rootdir: /Users/perry/src/cpppo_positioner
collected 2 items

../../Users/perry/src/cpppo_positioner/smc_test.py::test_smc_basic PASSED
../../Users/perry/src/cpppo_positioner/smc_test.py::test_smc_position PASSED
```

You'll also see the traffic in the terminal you started the `cpppo_positioner.ttyV-setup`.

If you have `ttyS0`, ... symbolic links in your current directory connected to USB RS-485 devices and they are wired together (GND, A+ and B- connected), you may substitute `SERIAL_TEST=ttyS` in the above unit test, and they should also pass.

1. Testing on Windows with USB-RS485 Devices

   Plug in 2 USB-RS485 devices, and start the simulator in a Powershell Terminal:

   ```
   PS> ./SMC-Project/Scripts/Activate.ps1
   (SMC-Project) PS> python -m cpppo_positioner.simulator -v --actuator 1 --address COM3
   Success; Started Modbus/RTU Simulator; PID = 5692; address = COM3
   ```

   In another Powershell Terminal:

   ```
   PS> ./SMC-Project/Scripts/Activate.ps1
   (SMC-Project) PS> python -m cpppo_positioner --address COM4 -vv \
       '[1,"RESET"]' 1 '[1,"reset"]' 1
   ```

   You should see I/O logging and the polled SMC Modbus/RTU Gateway status.

2. Windows Unit Testing

   To test on Windows, plug in 2 USB-RS485 adapters; they should be automatically provisioned as COM3 and COM4. Ensure that the RS-485 adapters are wired together directly: `A+-` to `A+`, `B-` to `B-`, and `GND` to `GND`.

   Run the unit tests w/ a normal level of logging:

   ```
   PS> winget install --id Git.Git -e --source winget  # (Re-open the Terminal to update $PATH)
   PS> ./SMC-Project/Scripts/Activate.ps1
   (SMC-Project) PS> git clone https://github.com/pjkundert/cpppo_positioner.git
   (SMC-Project) PS> cd cpppo_positioner
   (SMC-Project) PS> python -m pip install .[all,tests]
   (SMC-Project) PS> $env:SERIAL_TEST="COM"; python -m pytest -k position -v --capture=no --log-cli-level=25
   ...
   smc_test.py::test_smc_position
   --------------------------------------------------- live log call ------------------------------------------------
   NORMAL   root:smc_test.py:349 Using Actuator Simulator on COM4
   WARNING  root:pymodbus_fixes.py:120 Listen on server_listener...
   NORMAL   root:pymodbus_fixes.py:129 Communication established on server_listener
   NORMAL   root:smc.py:454 Position: actuator   1 updated:        position:       0 (== [0, 0])
   NORMAL   root:smc.py:454 Position: actuator   1 updated:   movement_mode:       1 (== [1])
   NORMAL   root:smc.py:454 Position: actuator   1 updated:           speed:     500 (== [500])
   NORMAL   root:smc.py:454 Position: actuator   1 updated:    acceleration:    5000 (== [5000])
   NORMAL   root:smc.py:454 Position: actuator   1 updated:    deceleration:    5000 (== [5000])
   NORMAL   root:smc.py:454 Position: actuator   1 updated:   pushing_force:       0 (== [0])
   NORMAL   root:smc.py:454 Position: actuator   1 updated:   trigger_level:       0 (== [0])
   NORMAL   root:smc.py:454 Position: actuator   1 updated:   pushing_speed:      20 (== [20])
   NORMAL   root:smc.py:454 Position: actuator   1 updated:    moving_force:     100 (== [100])
   NORMAL   root:smc.py:454 Position: actuator   1 updated:     in_position:     100 (== [0, 100])
   PASSED
   ```

1.4 Positioning

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

1.4.1 RS-485 I/O Device Setup

Your SMC actuator must be available via RS-485 from the computer. We assume that the actual underlying device is available via a symbolic link `ttyS0` in the current directory. For example, if this is a USB RS-485 interface, it might actually be `/dev/tty.usbserial-B0019I24`; identify this device file, go to the directory in which you are going to running the `cpppo_positioner` code, and run:

```
$ ln -fs /dev/tty.usbserial-B0019I24 ttyS0
```

Alternatively, specify the `address` parameter when calling `smc.smc_modbus()`.

1.4.2 `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()  # Assumes "ttyS0" is the Modbus device
```

keyword	description
address	The serial port device address, default "ttyS1"
timeout	The RS-485 I/O timeout, default .075s
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.

1.4.3 `.alarm` – Test for alarm condition

The .alarm returns the current ALARM condition (in **reverse** logic) if the device is available, or None:

Value	Description
None	No ALARM condition value available
0	ALARM condition is **Set**
!0	ALARM condition is **Clear**

```
alarm_value = gateway.alarm( actuator=1 )  # Detect ALARM value, and if Set then clear it
```

keyword	default	description
actuator	1	The actuator number to operate on
forget	True	Ignore the current stored value, and attempt to poll within timeout
reset	True	If ALARM condition found to be **Set**, clear it
timeout	None	Allowed number of seconds to complete (forever if None)

1.4.4 `.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	default	description
actuator	1	The actuator number to operate on
svoff	False	If positioning complete, turn off servo
noop	False	Don't return home, write new step data but don't initiate
timeout	None	Allowed number of seconds to complete (forever if None)

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 )
```

1.4.5 `.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	default	description
actuator	1	The actuator number to operate on
timeout	None	Allowed number of seconds to complete (forever if None)
svoff	False	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 )
```

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

Modifies one or more named outputs (Coils) on the specified actuator; a variable number of positional parameters:

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

```
gateway.outputs( "hold", "RESET", actuator=1 )  # clears HOLD, and sets RESET on SMC #1
```

keyword	default	description
*flags	[]	Positional parameters for each flag to "SET" or "reset"
actuator	1	The actuator number to operate on

1.4.7 `.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	default	description
actuator	1	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
}
```

1.4.8 `.close` – Terminate polling of serial port, close device

Ceases I/O to all actuators on RS485 circuit and releases the serial device.

1.4.9 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`, ...). Here's an example that works on Posix system shells (eg. bash on Linux, macOS):

```
$ 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):

```
$ ln -fs /dev/tty.usbserial-B0019I24 ttyS0  # or just use eg. COM3 on Windows
$ python -m cpppo_positioner --address ttyS0 -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 --address COM3 '[1,"RESET"]' 1 '[1,"reset"]' 1 \
   '{"actuator":1, "position":1000}' '{"actuator":1,"timeout":10}'
```

On Windows Powershell, this will be something like:

```
$ python -m cpppo_positioner -vv --address COM3 '[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!)

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 back-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…

2. Update on Windows Powershell Quoting

   Powershell appears to have updated its escape handling. Using the `$position` environment variable approach still doesn't work, but the following now works (oddly):

   ```
   python -m cpppo_positioner -v '{ \"actuator\": 0, \"position\": 12345, \"speed\": 100 }
   ```

2 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 cpppo_positioner, **or** are in the directory containing the cloned cpppo_positioner repository): To simulate an SMC positioning actuator 1 on ttyS1 (a symbolic link in the current directory to the actual RS-485 serial interface in /dev, eg. `/dev/tty.usbserial-B0019I24`):

```
$ python -m cpppo_positioner.simulator --address ttyS1 --actuator 1
```

2.1 Virtual Serial Devices

To run a simple Modbus/RTU simulator with some of the register addresses of SMC Actuators on a simulator virtual RS-485 network, you can use the `cpppo_positioner.ttyV-setup` script.

By default, this will create three virtual Pseudo-TTY devices in the current directory: `ttyV0`, `ttyV1` and `ttyV2`. All traffic to any of these devices will be written to all of them. This simulates a rudimentary multi-drop RS-485 network for testing.

Use `ttyV1`, ... as the target(s) for your `cpppo_positioner.simulator` server(s), instead of `ttyS1` or your actual RS-485 I/O device address.

Use `ttyV0` as the device to connect your `cpppo_positioner`

In a terminal, start the `ttyV` simulated serial device service like this:

```
$ . ./SMC-Project/bin/activate
(SMC-Project) $ python -m cpppo_positioner.ttyV-setup
ttyV0 -> /dev/ttys016
ttyV1 -> /dev/ttys017
ttyV2 -> /dev/ttys018
...
```

This will block, and when traffic begins to flow across the simulated `ttyV...` devices, you'll see the number of bytes and which `ttyV#` port it arrived at and was sent to:

```
41 <-- ttyV1: 0x0103240000000000000000000000000000000000000000000000000000000000000000000000
41 --> ttyV0
41 --> ttyV2
8 <-- ttyV0: 0x000200480001380d
8 --> ttyV1
8 --> ttyV2
```

2.2 Simulated Positioning

Once your simulated RS-485 network:

```
(SMC-Project) $ python -m cpppo_positioner.ttyV-setup
```

is up and your simulator:

```
(SMC-Project) $ cpppo_positioner.simulator --address ttyV1 --actuator 1
```

is running, try to write to some registers with delays of 1 second between each:

```
(SMC-Project) $ python -m cpppo_positioner --address ttyV0 -vvv \
    '[1,"RESET"]' 1 '[1,"reset"]' 1
```

and send a positioning request:

```
(SMC-Project) $ python -m cpppo_positioner --address ttyV0 -vvv \
    '{ "actuator": 0, "position": 12345, "speed": 100 }'
```

You can use this to process a complex stream of commands, taking a stream of commands from standard input where the - is in the list, and then going to the final position after completing the stream of requests:

```
(SMC-Project) $ python -m cpppo_positioner --address ttyV0 -vvv \
    '{ <initial position> }' '# a comment, followed by a delay' 1.5 - '{ <final position> }'
```