Serial Mock Documentation

Release 0.0.1a1

Joran Beasley

Feb 01, 2019

Contents

I wrote SerialMock in order to make testing interfaces with devices a bit easier. I felt like there was no good existing solution to this problem you can install it with any one of

CHAPTER 1

Requirements

- Python2.7
- a serial port to bind to (whether it is a hardware null modem or software should not matter).
  - *in general you can use com0com for windows systems, and socat for *nix systems in order to create a software null modem*

See also:

1.1 Setting up a Null Modem

1.1.1 Windows : com0com

Download one of the two versions below (I recommend using the signed version)

- Download official unsigned com0com
  - *there is a readme with directions on how to self-sign the driver*
- Download signed com0com 2.2.0 drivers (recommended)

In windows you need to create a null modem serial port.

the easiest (read cheapest) way to do this is using com0com, however there are many other options out there, simply google "windows null modem software", and you should be able to find many alternatives.

1.1.2 Linux : socat

Install socat with `apt-get install socat`

```
apt-get install socat
socat -d -d pty,raw,echo=0 pty,raw,echo=0
sudo ln -s /dev/pts/## /dev/ttyUSB0
```

CHAPTER 2

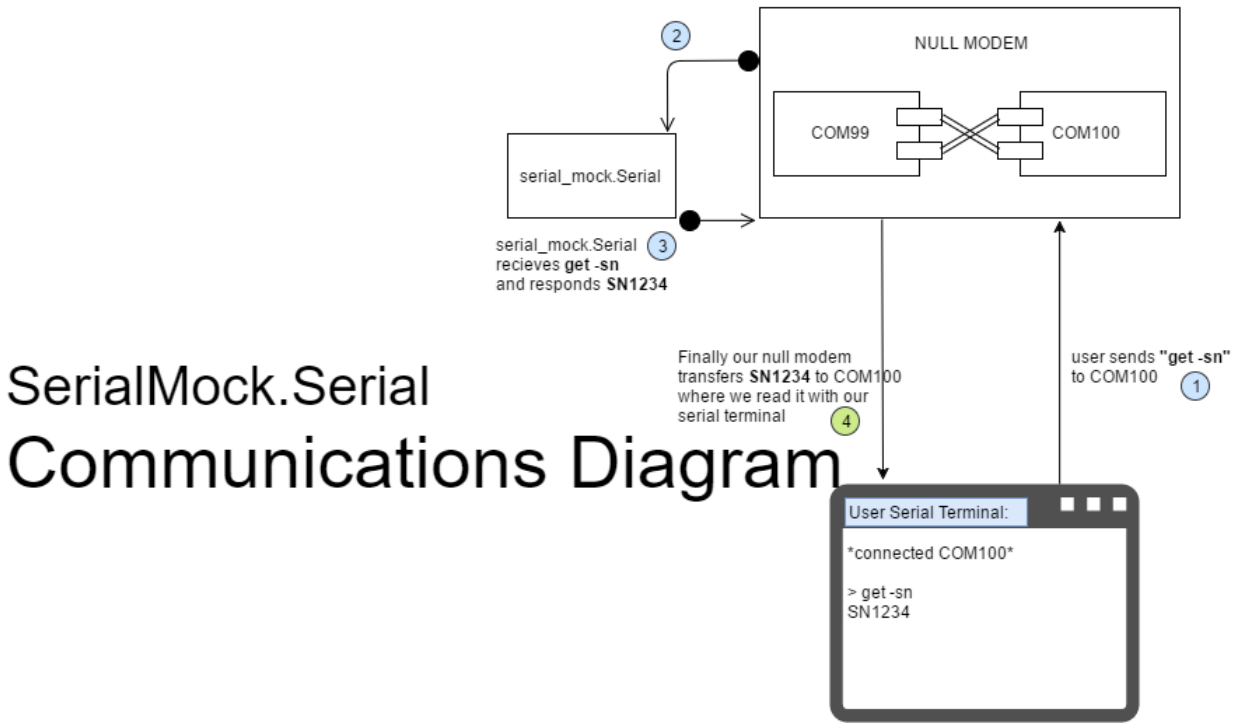
Installation

`setup.py install`
or `pip install .`
or install it from pipy with `pip install serial_mock`
or directly from github `pip install git+https://github.com/joranbeasley/SerialMock.git`

CHAPTER 3

API Reference

3.1 serial_mock API



3.1.1 MockSerial Class

class serial_mock.**MockSerial**(*stream*, *logfile=None*, ***kwargs*)

MockSerial(stream:string) instanciates a new MockStreamTunnel, stream should point to the comm port to

listen on. *in general this class should not be directly invoked but should be subclassed, you can find some examples in the examples folder, or in the cli.py file*

Parameters

- **stream** – a path to a pipe (ie "/dev/ttyS99","COM11"), a stream like object, or "DEBUG"
- **data_prefix** – the separator between getters/setters and the data_attribute they reference

```
>>> from serial_mock.decorators import serial_query
>>> from serial_mock.mock import MockSerial
>>> class SimpleSerial(MockSerial):
...     simple_queries = {
...         "get -name":"hello my name is bob",
...         "get -next":["123","456","789"],
...         "get -id":12
...     }
...     data={"x":6}
...     @serial_query("trigger command")
...     def do_something(self,requiredArg,optionalArg="0"):
...         return "RESULT: %r %r"%(requiredArg,optionalArg)
...
>>> mock = SimpleSerial("DEBUG")
>>> mock.process_cmd("trigger command 1")
"RESULT: '1' '0'"
>>> mock.process_cmd("trigger command 1 2")
"RESULT: '1' '2'"
>>> mock.process_cmd("get -name")
'hello my name is bob'
>>> mock.process_cmd("get -next")
'123'
>>> mock.process_cmd("get -next")
'456'
>>> mock.process_cmd("get -next")
'789'
>>> mock.process_cmd("get -next")
'123'
>>> mock.process_cmd("get -id")
'12'
>>> mock.process_cmd("get -x")
'6'
>>> mock.process_cmd('set -x 10')
'OK'
>>> mock.process_cmd("get -x")
'10'
```

data = {}
any keys defined in **data** will automatically have getters or setters created for them

data_prefix = '-'
the prefix to use with data auto generated routes

baudrate = 9600
the **baudrate** we should operate at

prompt = '>'
the **prompt** to display to the user

delimiter = '\r'
user_terminal defines the character(or characters, or regexp, or list) of items that indicate our user has finished a command

endline = '\r'
endline defines the character to output after our response but before our prompt

simple_queries = {}
any key:value pair in simple queries is exposed as a simple query response ... and the query string must be an exact match value can be a string/unicode/bytes value or it can be a list or array, if a list or array is passed in then the responses will be cycled any other value type will be coerced to *str*

logfile = None

process_cmd(*cmd*)
looks up a command to see if its registered. and returns the result if it is otherwise returns an error string in general this command should not be invoked directly (but it can be...)

```
>>> from serial_mock.mock import MockSerial
>>> inst = MockSerial("DEBUG")
>>> inst.process_cmd("a")
"ERROR 'a' Not Found"
```

Parameters **cmd** – the command to process

Returns a string (the result of the command)

terminate()
stop the MainLoop if running :return:

MainLoop()
Mainloop will run forever serving the rules provided in the subclass to the bound pipe

3.1.2 DummySerial Class

class serial_mock.**DummySerial**(*MockSerialClass*)
DummySerial provides a serial.Serial interface into a MockSerial instance. you can use this as a dropin replacement to serial.Serial, for anything that accepts serial.Serial as an argument

```
>>> from serial_mock.mock import DummySerial,MockSerial
>>> from serial_mock.decorators import serial_query
>>> class MyInterface(MockSerial):
...     @serial_query("trigger command")
...     def do_something(self,requiredArg,optionalArg="0"):
...         return "RESULT: %r %r"%(requiredArg,optionalArg)
...
>>> ser = DummySerial(MyInterface)
>>> ser.write("trigger command 5\r")
18L
>>> ser.read(ser.inWaiting())
"RESULT: '5' '0'\r>"
>>> ser.write("trigger command 1 2\r")
20L
>>> ser.read(ser.inWaiting())
"RESULT: '1' '2'\r>"
```

is_open = True

open()
Open port with current settings. This may throw a SerialException if the port cannot be opened.

close()
Close port

in_waiting

inWaiting()

write(*msg*)
Output the given byte string over the serial port.

read(*bytes=1*)
Read size bytes from the serial port. If a timeout is set it may return less characters as requested. With no timeout it will block until the requested number of bytes is read.

3.1.3 EmittingSerial Class

class serial_mock.**EmittingSerial**(*stream*, *logfile=None*, ***kwargs*)
EmmitingSerial(stream:string) provides a reference class on an interface that periodically emits a "heartbeat" type message

Parameters

- **stream** – a path to a pipe (ie "/dev/ttyS99","COM11"), a stream like object, or "DEBUG"
- **data_prefix** – the separator between getters/setters and the data_attribute they reference

emit = 'EMIT MSG'

delay = (5, 35)

interval = (15, 35)

MainLoop()
Mainloop will run forever serving the rules provided in the subclass to the bound pipe

3.1.4 debug output

these modules make use of pythons logging module, you can set the verbosity with `logging.getLogger("serial_mock").setLevel(logging.DEBUG)`

Indices and tables

- genindex
- search

3.2 serial_mock *serial_query* Decorator

@serial_mock.**serial_query**(*route=None*, *delay=None*)

Parameters

- **route** (*str*) – the serial instruction to recieve
- **delay** (*int or float*) – How long this command takes to return

specify a class function as a serial interface... if you do not specify a route, it will default to a "normalized" version of the
the method MUST accept at least one argument, the instance of the `serial_mock.mock.Serial` *that is being run.*

3.2.1 the route argument

The **route** argument is the primary argument to this decorator and it is very flexible. by default it will convert the function name into a "serial query"

default behavior

for most use cases the default behavior should be sufficient to meet your needs, if it doesnt, have no fear, explicitly declaring the route gives you near unlimmited flexibility

```
>>> @serial_query
... def show(self):
...     return "Info to show"
```

in this instance the route manager exposes the serial command "show" to this method.

```
>>> @serial_query
... def get_sn(self):
...     return self.sn
...
>>> @serial_query
... def set_sn(self,new_sn):
...     self.sn = new_sn
...     return "OK" # in general you always want serial queries to respond with some␣
↪data
```

in the above example we expose 2 new routes, we expose "get sn" which accepts no additional data, and also a "set sn" which expects one addition argument of the new serial number, it would be triggered with a command like "set sn SN123123" this is effectively what happens with any variables defined in your `serial_mock.Serial` subclass' `data` attribute

you could also accept multiple arguments

```
>>> @serial_query
... def set_usercal(self,offset,slope=0)
...     return "OK"
```

in this example this method would be invoced with "set usercal 4" or "set usercal 4 6", allowing you to optionally pass in a second variable, you could of coarse require the second variable or 3 variables, etc.

explicitly declared routes

perhaps you are emulating a device that has commands that are not part of legal function names in python, consider something like "#00x53"

```
>>> @serial_query("#00x53")
... def show_info():
...     return "blah i am info"
```

in this example the user can pass "#00x53" to the serial port and it will trigger this method., anything that follows will be split on spaces and passed in as arguments

complex routes

you can also pass in a compiled regex to match against ... any groups will be passed as arguments to function that is bound to this trigger

```
>>> @serial_query(re.compile("(.*)"))
... def echo_function(self,user_msg):
        return user_msg
```

this is a regex that will match anything and pass it into this function, of coarse you can use much more complex regular expressions, though you rarely need to.

3.2.2 Examples

```
from serial_mock import Serial
from serial_mock import serial_query

class MySerial(Serial):
    ...
    @serial_query # since we did not specify an explicit route,this will default to a␣
↪rout of "get name"
    def get_name(self):
         return self.name

    @serial_query # again this will default to "set name" and will expect one␣
↪argument (the name to set)
    def set_name(self,name):
         self.name = name
         return "OK"

    @serial_query("quick scan") # this time we will override the command, if we did␣
↪not the route would be "do scan"
    def do_scan():
        return " ".join(map(str,range(9)))

    @serial_query("long scan",delay=5) # this time we will do a long scan with a␣
↪delay of 5 seconds
    def do_long_scan(): # the decorator will take care of the delay for us
        return self.do_scan() # note that the decorator leaves the original function␣
↪unaffected
```

3.3 serial_mock *bind_key_down* Decorator

the `@bind_key_down` decorator allows you to bind a function to a keypress, this can be usefull to perform sporatic actions (like incrementing an id)

```
class MyInterface(SerialMock):
     current_id = 1
     @serial_query("get -record_id")
     def get_id(self):
         return "%s"%self.current_id

     @bind_key_down("a")
```

(continues on next page)

(continued from previous page)

```
    def increment_id(self):
        self.current_id += 1
```

in this example when the user presses 'a' the current_id attribute will increase by one. and the next time "get - record_id" is invoked the new current_id is returned to the client.

3.3.1 Indices and tables

- genindex
- search

CHAPTER 4

Tools

4.1 Serial Terminal Programs

there are several utilities available for different os's. some of the big ones are

4.1.1 Windows Options

- TeraTerm (more complex, lots of options... doesnt handle `\r` linefeeds very well)
- DecaTerm (recommended if your device mostly communicates in ascii, very easy to use)

4.1.2 Linux Options

- `screen` is a built in pts communication application that works pretty well
  - `screen /dev/ttyS0 9600` would open up a terminal to /dev/ttyS0 device at 9600 baud
- `cu` is another built in nix utility for communicating with a subshell
  - `cu -l /dev/ttyS0 -s 9600` would open up a terminal to /dev/ttyS0 device at 9600 baud
- `minicom` is a more robust application designed for port communication in linux. invoke with `minicom` and follow on screen menu system
- CuteCom a graphical cross platform serial terminal

4.2 Using the include CLI tool

In order to use the CLI tool, you must first create a null modem.

to execute the cli command simply invoke

```
python -m serial_mock.cli COMMAND`
```

the available commands are detailed below

4.2.1 CLI commands

to view help on a given command simply invoke the command with the `-h` or `--help` switch

you can optionally specify a verbosity level for screen output with `-v <LOGLEVEL>` **BEFORE** you invoke the {echo,bridge,build} command, this can be helpful for debugging

echo directive

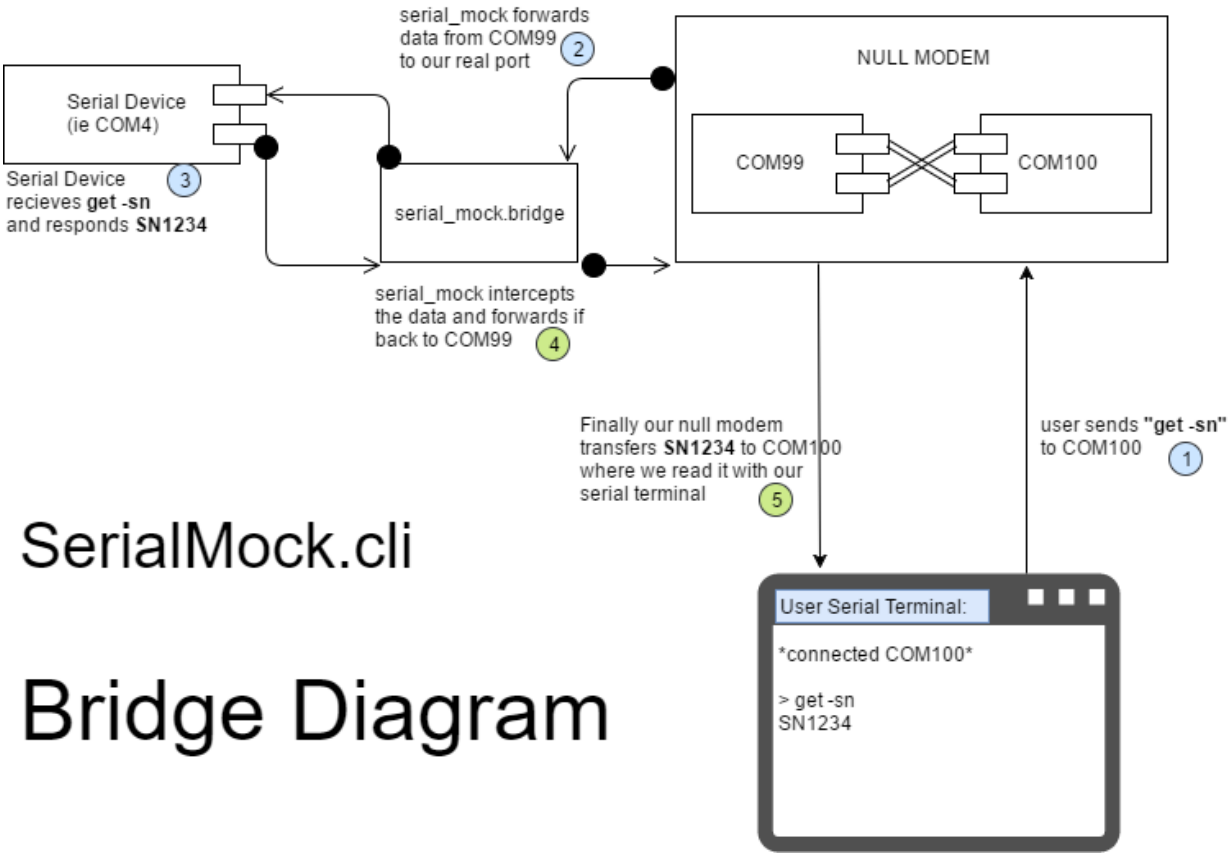
the `echo` cli directive will bind a simple echo device to the specified bind port

```
python -m serial_mock.cli echo COMPORT
```

where **COMPORT** is one half of a pair of ports specified when you created a null modem.

bridge directive

the `bridge` cli directive will create a bridge between two points, optionally capturing the traffic to a log file for use with the `build` directive



```
python -m serial_mock.cli bridge COMPORT1 COMPORT2 <options>
```

where **COMPORT1** and **COMPORT2** are the two ports you wish to bridge, typically one port will be an actual connected device and the 2nd port will be one end of a null modem that you created.

See also:

build directive

the build directive can convert a logfile created with the `bridge` directive, and convert it into a "playback" device, that will play back the responses from the bridged session

```
python -m serial_mock.cli build serial_output.txt --out=MySerialDevice.py
```

you can then serve your mocked serial port with

```
python MySerialDevice.py COM100
```

this will bind your mocked device to COM100, which will expose it at the other end of the null modem (COM99)

- genindex
- search

CHAPTER 5

Examples

5.1 Tutorial

5.1.1 Getting Started

We will go ahead and mock a complete instrument in this tutorial. we will start by "cloning" our existing device using the *cli_util*

PreRequisites

you must have already created a null modem and have taken notes of the names of both endpoints. I have chosen COM99 and COM100, as my two endpoints using windows.

Note: in linux your endpoints should look more like `/dev/ttyS0`

See also:

"Cloning" an existing device

1. connect your device to a comport and make note of its identifier (something like `COM2` in windows and `/dev/ttyUSB0` in unix-like systems.
2. run the following cli command `python -m serial_mock.cli bridge COM2 COM99 -L output_file_name.data` this will create a bridge between COM2 and COM99, in this case COM2 is our device and COM99 is one end of our null modem
3. now connect up to the other end of our null modem (I am using COM100) (remember i bound to COM99 in the above command and my null modem pair is COM99 <-> COM100), using a serial terminal program of your choice

See also:

4. once connected send a series of commands you would like to clone, the input and output will be recorded into our `output_file_name.data` file that we specified before
5. once you are done simple exit our command line cli instruction with `ctrl+c`, we now have our logfile that will serve as the foundation of our cloned device
6. to convert it into a serial_mock object we will simply use our cli command to `build` our instance, with `python -m serial_mock.cli bridge output_file_name.data --out=MySer.py`

- this will create a new file MySer.py that we can run to mimic our recorded device

7. Finally serve up our mocked device with `python MySer.py COM99` which will serve our mocked port on the other half of the null modem (ie. connect to COM100 to interact with it)

5.2 EXAMPLES

5.2.1 simple gps example

```
from serial_mock import Serial,serial_query

class GPSSerial(Serial):
     baudrate=115200
     position={'x':1,'y':2,'z':3}
     @serial_query("get -p")
     def get_position():
          return struct.pack("BBB",[position['x'],position['y'],position['z']])


GPSSerial("COM99").MainLoop()
```

5.2.2 real life example: Omega PH METER

http://www.omega.com/manuals/manualpdf/M4278.pdf

```
from serial_mock import Serial,serial_query

class Omega_PHH37(Serial):
     baudrate=115200
     user_terminal="\r\n"
     prompt=""
     reading={'ph':7.2,'status':'OK','mv':3.1}
     @serial_query("#001N")
     def get_reading(self):
          return "\xff\xfe\x02\x06\x06"+"{status}{ph:0.4f}{mv:0.4f}\xaa\xbb".
↪format(**self.reading)

 Omega_PHH37("COM99").MainLoop()
```

5.2.3 Indices and tables

- genindex

- search

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Index