driver2200087 Documentation

Release 0.6

Chintalagiri Shashank

August 19, 2015

Contents

CHAPTER 1

driver2200087

This is a Python Package to interface with Radio Shack's 2200087 Multimeter.

The 2200087 is an inexpensive DMM which supports logging and graphing data on a computer, but the supplied code only supports Windows. This is a python package to allow for connecting to the multimeter over USB on Linux or Mac OSX. Due to radioshack not supplying any serial specifications, the protocol was reverse engineered by David Dworken from simply observing the output of the DMM.

The `serialDecoder` module and the serial protocol documentation is essentially a slightly tweaked version of the script written and maintained by David Dworken, available at https://github.com/ddworken/2200087-Serial-Protocol.git

This package includes a version of the `serialDecoder` module, slightly refactored to allow it to be imported into other python scripts. It also includes a `runner` module which contains a Twisted protocol, wrapped by Crochet. This module should be relatively easier to include into other python scripts and applications.

1.1 Installation

This package has been tested only with python 2.7.

This package can be installed from pypi using pip:

```
$ pip install driver2200087
```

Or using easy_install (python 2.7 only):

```
$ easy_install driver2200087
```

1.2 Usage

Standalone usage is listed in the documentation, and should be fairly straightforward to follow.

The `serialDecoder` module can also be imported and used from within a python script, and the documentation can help you use it in that manner.

The recommended way for using the package, though, is through the `runner` module which it provides. The simpler form of use is to get the latest available value whenever necessary. A short example of how this can be done using this package in a typical python application would be

```
from driver2200087 import runner

dmm = runner.InstInterface2200087()
dmm.connect()

# Other Application code
# ...
#
# And when the measurement is required :

if dmm.data_available() > 0:
    print dmm.latest_point()
else:
    raise Exception # Or pass, or retry, as per application requirements

# other application code
# ...
```

If the application calls for continuous recording of the data, the following is likely a better approach

```
from driver2200087 import runner

dmm = runner.InstInterface2200087()
dmm.connect()
while True:
    if dmm.data_available() > 0:
        print dmm.next_point()
```

Note that in this code snippet, the python interpreter is blocked by the infinite while loop. This is not required by the module. As long as dmm.next_point() is called often enough (10 Hz), you can use whatever mechanism you like to actually make the call. Note that dmm.data_available() **must** be checked by your application before making the call, or you should trap the exception that results.

Making the call at less than this frequency will cause data points to be lost when the point buffer fills up - if your application only calls for the occasional measurement, you're probably better off with dmm.latest_point() instead.

For an example of using the `runner` module from within a larger framework by subclassing the provided twisted protocol, see the corresponding Tendril module. The `Tendril` module, while WIP, also includes examples of parsing the obtained datapoint strings into usable values.

1.3 Downloads and Documentation

The simplest way to obtain the source for this package is to clone the git repository:

```
git clone https://github.com/chintal/driver2200087.git driver2200087
```

You can install it as usual, with:

```
python setup.py install
```

The latest version of the documentation can be found at ReadTheDocs.

1.4 License

driver2200087 is distributed under the GPLv2 license.

Contents:

1.4.1 Standalone Usage

Start by cloning this repository:

```
git clone https://github.com/chintal/driver2200087.git driver2200087
```

or the original by David Dworken:

```
git clone https://github.com/ddworken/2200087-Serial-Protocol.git
```

Then install dependencies:

```
pip install numpy pyserial
```

Then you're ready to go. Just run the program to display a text output of the data:

```
sudo python serialDecoder.py -p /dev/ttyUSB0
```

If you want a graph as your output, first install GNUPlot:

```
sudo apt-get install gnuplot
```

then run:

```
sudo python serialDecoder.py -p /dev/ttyUSB0 --graph
```

You also can read from multiple multimeters at the same time and get a CSV output like so:

```
sudo python serialDecoder.py -p /dev/ttyUSB0 /dev/ttyUSB1
```

If you only want the actual values and not information about what mode the multimeter is on, use the -q flag:

```
sudo python serialDecoder.py -p /dev/ttyUSB0 -q
```

1.4.2 2200087 Serial Protocol Description

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Byte 1	0	0	0	1	Minus	AC	SEND	AUTO
Byte 2	0	0	1	0	Continuity	Diode	Low Batt	Hold
Byte 3	0	0	1	1	MAX	E4	F4	A4
Byte 4	0	1	0	0	D4	C4	G4	B4
Byte 5	0	1	0	1	DP3	E3	F3	A3
Byte 6	0	1	1	0	D3	C3	G3	B3
Byte 7	0	1	1	1	DP2	E2	F2	A2
Byte 8	1	0	0	0	D2	C2	G2	B2
Byte 9	1	0	0	1	DP1	E1	F1	A1
Byte 10	1	0	1	0	D1	C1	G1	B1
Byte 11	1	0	1	1	Percent	HFE	Rel Delta	MIN
Byte 12	1	1	0	0	u (1e-6)	n (1e-9)	dBm	Seconds
Byte 13	1	1	0	1	Farads	Amps	Volts	m (1e-3)
Byte 14	1	1	1	0	Hz	Ohms	K (1e3)	M (1e6)

All bytes are sent over in hexadecimal numbered one through fourteen. Bytes 3-4 contain digit 4, bytes 5-6 contain digit 3 and so on. All other parts of the display are turned on as shown in the above table. The multimeter sends the data at a rate of 10 Hz.

1.4.3 driver2200087 package

driver2200087.serialDecoder module

Serial Decoder for RadioShack 2200087 Multimeter

This module provides functions for decoding the serial protocol of the RadioShack 2200087 Mulitimeter. See the included *basic.rst* for the protocol specifications and standalone usage instructions for the script.

The documentation in this file focuses on the usage of this file as a module.

driver2200087.serialDecoder.**detect_device_port**()

Locate the RS2200087 multimeter using whatever information is available. Specifically, look for :

- Prolific 2303 serial ports
- Which produce data that can be parsed by this module

As long as no other connected USB devices have a Prolific 2303 serial port, it should be fine. This is an insufficient test, and should be avoided in favor of manually specifying the port. This is especially true when other USB devices containing Prolific 2303 serial ports are also expected to be connected.

class driver2200087.serialDecoder.**Grapher**(*y*)

Bases: object

Grapher used to plot a graph of the data when used in standalone mode. When used as a module, you can probably just ignore it and use your own graphing mechanism, if any.

np = <module 'numpy' from '/usr/lib/python2.7/dist-packages/numpy/__init__.pyc'>

subprocess = <module 'subprocess' from '/usr/lib/python2.7/subprocess.pyc'>

x = []

graphSize = 100

y = []

graphOutput = []

update(*x*, *y*, *label='DMM'*)

get_graph()

get_values()

append(*y_val*)

append_with_label(*y_val*, *label*)

driver2200087.serialDecoder.**get_arr_from_str**(*serial_data*)

Converts serial data to an array of strings each of which is a binary representation of a single byte

Parameters **serial_data** (*str*) – Series of bytes received over the serial line, separated by spaces

Returns list of ascii representations for each character in the serial data

Return type list

driver2200087.serialDecoder.**process_digit**(*digit_number*, *bin_array*)

Extracts a single digit from the binary array, at the location specified by *digit_number*, and returns it's numeric value as well as whether a decimal point is to be included.

Parameters

- **digit_number** (*int*) – Location from which digit should be extracted (4, 3, 2, 1)

- **bin_array** (*list*) – Array of binary representations of serial data

Return type tuple

Returns decimal_point_bool Boolean if decimal point is to be included at the specified location

Returns digit_value Number value of the digit at the specified location

driver2200087.serialDecoder.**get_char_from_digit_dict**(*digit_dict*)
Converts a digit_dict into the character it represents.

Parameters **digit_dict** (*dict*) – dictionary containing the digit's information

Returns The character represented by digit_dict

Return type int or char

driver2200087.serialDecoder.**is_e**(*digit_dict*)

driver2200087.serialDecoder.**is_n**(*digit_dict*)

driver2200087.serialDecoder.**is_l**(*digit_dict*)

driver2200087.serialDecoder.**is_p**(*digit_dict*)

driver2200087.serialDecoder.**is_f**(*digit_dict*)

driver2200087.serialDecoder.**is_c**(*digit_dict*)

driver2200087.serialDecoder.**is_9**(*digit_dict*)

driver2200087.serialDecoder.**is_8**(*digit_dict*)

driver2200087.serialDecoder.**is_7**(*digit_dict*)

driver2200087.serialDecoder.**is_6**(*digit_dict*)

driver2200087.serialDecoder.**is_5**(*digit_dict*)

driver2200087.serialDecoder.**is_4**(*digit_dict*)

driver2200087.serialDecoder.**is_3**(*digit_dict*)

driver2200087.serialDecoder.**is_2**(*digit_dict*)

driver2200087.serialDecoder.**is_1**(*digit_dict*)

driver2200087.serialDecoder.**is_0**(*digit_dict*)

driver2200087.serialDecoder.**str_to_flags**(*str_of_bytes*)
Checks all possible flags that might be needed and returns a list containing all currently active flags

Parameters **str_of_bytes** (*str*) – a string of bytes

Returns list of flags, each of which is a string

Return type list

driver2200087.serialDecoder.**str_to_digits**(*str_of_bytes*)
Converts a string of space separated hexadecimal bytes into numbers following the protocol in readme.md

Parameters **str_of_bytes** (*str*) – a string of bytes

Return type str

Returns string of digits represented by str_of_bytes with decimal point as applicable

driver2200087.serialDecoder.**get_serial_chunk**(*ser*)
Gets a serial chunk from the device.

Parameters **ser** – serial.Serial object

Return type str

Returns string of 14 received characters separated by spaces

driver2200087.serialDecoder.**process_chunk**(*chunk*)

driver2200087.serialDecoder.**get_next_point**(*ser*)

Get the next point from the device. This function raises an Exception if anything at all goes wrong during the process of obtaining the value. The returned value is a string which should then be parsed by downstream code to determine what it actually is.

Due to the nature of the serial interface, the downstream code must also ensure that this function is called often enough to keep the data in the various serial buffers from going stale. This particular DMM sends back a point every 0.1s, so this function should effectively be called at that frequency.

Alternatively, a crochet / twisted based protocol implementation can be used to provide an interface friendlier to more complex synchronous code without needing to create a plethora of threads that spend their time in time.sleep().

Warning: This function will block.

driver2200087.serialDecoder.**confirm_device**(*ser*)

Test the serial object for the device. This is a naive test, assuming that if a value can be successfully parsed, the device is what is expected. This is a very weak test, and should not be overly relied upon.

driver2200087.serialDecoder.**get_serial_object**(*port=None*)

Get a serial object given the port.

driver2200087.serialDecoder.**main_loop**(*vargs*)

Main loop for standalone use

driver2200087.runner module

This module provides an asynchronous backend to the RadioShack 2200087 multimeter's PC interface. It uses crochet to provide a synchronous API to an underlying Twisted based implementation.

While the intent of this module is to allow the use of the device from within a larger framework, the use of crochet should allow the use of this API and therefore the instrument in a naive python script as well.

See the 'main' section of this file for a minimal example of it's usage.

driver2200087.runner.**unwrap_failures**(*err*)

Takes nested failures and flattens the nodes into a list. The branches are discarded.

class driver2200087.runner.**InstProtocol2200087**(*port*, *buffer_size=100*)

Bases: twisted.internet.protocol.Protocol

This is a twisted protocol which handles serial communications with 2200087 multimeters. This protocol exists and operates within the context of a twisted reactor. Applications themselves built on twisted should be able to simply import this protocol (or its factory).

If you would like the protocol to produce datapoints in a different format, this protocol should be sub-classed in order to do so. The changes necessary would likely begin in this class's frame_recieved() function.

Synchronous / non-twisted applications should use the InstInterface2200087 class instead. The InstInterface2200087 class accepts a parameter to specify which protocol factory to use, in case you intend to subclass this protocol.

Parameters

- **port** (*str*) – Port on which the device is connected. Default '/dev/ttyUSB0'.
- **buffer_size** (*int*) – Length of the point buffer in the protocol. Default 100.

reset_buffer()
Resets the point buffer. Any data presently within it will be lost.

make_serial_connection()
Creates the serial connection to the port specified by the instance's _serial_port variable and sets the instance's _serial_transport variable to the twisted.internet.serialport.SerialPort instance.

break_serial_connection()
Calls loseConnection() on the instance's _serial_transport object.

connectionMade()
This function is called by twisted when a connection to the serial transport is successfully opened.

connectionLost(*reason=<twisted.python.failure.Failure <class 'twisted.internet.error.ConnectionDone'>>*)
This function is called by twisted when the connection to the serial transport is lost.

dataReceived(*data*)
This function is called by twisted when new bytes are received by the serial transport.

This data is appended to the protocol's framing buffer, _buffer, and when the length of the buffer is longer than the frame size, that many bytes are pulled out of the start of the buffer and frame_recieved is called with the frame.

This function also performs the initial frame synchronization by dumping any bytes in the beginning of the buffer which aren't the first byte of the frame. In its steady state, the protocol framing buffer will always have the beginning of a frame as the first element.

Parameters **data** (*str*) – The data bytes received

frame_received(*frame*)
This function is called by data_received when a full frame is received by the serial transport and the protocol.

This function recasts the frame into the format used by the serialDecoder and then uses that module to process the frame into the final string. This string is then appended to the protocol's point buffer.

This string is treated as a fully processed datapoint for the purposes of this module.

Parameters **frame** (*str*) – The full frame representing a single data point

latest_point(*flush=True*)
This function can be called to obtain the latest data point from the protocol's point buffer. The intended use of this function is to allow random reads from the DMM. Such a typical application will want to discard all the older data points (including the one returned), which it can do with flush=True.

This function should only be called when there is data already in the protocol buffer, which can be determined using data_available().

This is a twisted protocol function, and should not be called directly by synchronous / non-twisted code. Instead, its counterpart in the InstInterface object should be used.

Parameters **flush** (*bool*) – Whether to flush all the older data points.

Returns Latest Data Point as processed by the serialDecoder

Return type str

next_point()
This function can be called to obtain the next data point from the protocol's point buffer. The intended

use of this function is to allow continuous streaming reads from the DMM. Such a typical application will want to pop the element from the left of the point buffer, which is what this function does.

This function should only be called when there is data already in the protocol buffer, which can be determined using data_available().

This is a twisted protocol function, and should not be called directly by synchronous / non-twisted code. Instead, its counterpart in the InstInterface object should be used.

Returns Next Data Point in the point buffer as processed by the serialDecoder

Return type str

next_chunk()

This function can be called to obtain a copy of the protocol's point buffer with all but the latest point in protocol's point buffer. The intended use of this function is to allow continuous streaming reads from the DMM. Such a typical application will want to pop the elements from the left of the point buffer, which is what this function does.

This function should only be called when there is data already in the protocol buffer, which can be determined using data_available().

This is a twisted protocol function, and should not be called directly by synchronous / non-twisted code. Instead, its counterpart in the InstInterface object should be used.

Returns Copy of point_buffer with all but the latest_point

Return type deque

data_available()

This function can be called to read the number of data points waiting in the protocol's point buffer.

This is a twisted protocol function, and should not be called directly by synchronous / non-twisted code. Instead, its counterpart in the InstInterface object should be used.

Returns Number of points waiting in the protocol's point buffer

Return type int

class driver2200087.runner.**InstFactory2200087**

Bases: twisted.internet.protocol.Factory

This is a twisted protocol factory which produces twisted protocol objects which handle serial communications with 2200087 multimeters. This class is typically not to be instantiated by application code. This module includes a single instance of this class (factory), which can be used to create as many such objects as are necessary.

This protocol factory exists and operates within the context of a twisted reactor. Applications themselves built on twisted should be able to simply import this protocol factory. Synchronous / non-twisted applications should use the InstInterface2200087 class instead.

buildProtocol(*port*, *buffer_size=100*)

This function returns a InstProtocol2200087 instance, bound to the port specified by the param port.

This is a twisted protocol factory function, and should not be called directly by synchronous / non-twisted code. The InstInterface2200087 class should be instantiated instead.

Parameters

- **port** (*str*) – Serial port identifier to which the device is connected
- **buffer_size** (*int*) – Length of the point buffer in the protocol. Default 100.

class driver2200087.runner.**InstInterface2200087**(*port=None*, *buffer_size=100*, *pfactory=<driver2200087.runner.InstFactory2200087 instance>*)

Bases: object

This class provides an synchronous / non-twisted interface to 2200087 multimeters. It uses the underlying _protocol object which does most of the heavy lifting using twisted / crochet.

For each DMM you want to connect to, instantiate this class once with the correct serial port string.

If you would like to use a custom protocol to interface with the device, you can do so by passing in the custom protocol factory as the named parameter pfactory. See the documentation of the default protocol object for information on creating a custom Protocol class.

Parameters

- **port** (*str*) – Port on which the device is connected. Default '/dev/ttyUSB0'.
- **buffer_size** (*int*) – Length of the point buffer in the protocol. Default 100.
- **pfactory** (InstFactory2200087) – Custom protocol factory to use, if not the one implemented here.

Your application code is expected to setup crochet before creating the instance. A short example :

```
>>> from crochet import setup
>>> setup()
>>> from driver2200087.runner import InstInterface2200087
>>> dmm = InstInterface2200087('/dev/ttyUSB0')
>>> dmm.connect()
>>> print dmm.latest_point()
```

connect(**args*, ***kwargs*)

This function connects to the serial port specified during the instantiation of the class.

This function should be called before anything else can be done with the object.

disconnect(**args*, ***kwargs*)

This function disconnects from the serial port specified during the instantiation of the class.

latest_point(**args*, ***kwargs*)

This function can be called to obtain the latest data point from the protocol's point buffer. The intended use of this function is to allow random reads from the DMM. Such a typical application will want to discard all the older data points (including the one returned), which it can do with flush=True.

This function should only be called when there is data already in the protocol buffer, which can be determined using data_available().

Parameters **flush** (*bool*) – Whether to flush all the older data points.

Returns Latest Data Point as processed by the protocol

Return type str or type of each datapoint

next_point(**args*, ***kwargs*)

This function can be called to obtain the next data point from the protocol's point buffer. The intended use of this function is to allow continuous streaming reads from the DMM. Such a typical application will want to pop the element from the left of the point buffer, which is what this function does.

This function should only be called when there is data already in the protocol buffer, which can be determined using data_available().

Returns Next Data Point in the point buffer as processed by the protocol

Return type str or type of each datapoint

next_chunk(**args*, ***kwargs*)
This function can be called to obtain the next chunk of data from the protocol's point buffer. The intended use of this function is to allow continuous streaming reads from the DMM. Such a typical application will want to pop the elements from the left of the point buffer, which is what this function effectively does.

This function should only be called when there is data already in the protocol buffer, which can be determined using data_available().

Returns Point buffer with all but the latest point in the protocol's point buffer

Return type deque or type of the point_buffer

data_available(**args*, ***kwargs*)
This function can be called to read the number of data points waiting in the protocol's point buffer.

Returns Number of points waiting in the protocol's point buffer

Return type int

reset_buffer(**args*, ***kwargs*)
This function can be called to reset the point buffer. This should be used for starting wave acquisition.

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

d

A

B

C

D

F

G

I