API reference for ssmdevices

Release 0.14

Dan Kuester, Paul Blanchard, Alex Curtin,
Keith Forsyth, Ryan Jacobs, John Ladbury,
Yao Ma, Duncan McGillivray, Andre Rosete,
Audrey Puls, Michael Voecks

Jun 16, 2023

CONTENTS

ssmdevices is a collection of python wrappers that have been used for automated experiments by the NIST Spectrum Technology and Research Division. They are released here for transparency, for re-use of the drivers ``as-is" by the test community, and as a demonstration of lab automation based on labbench.

The equipment includes consumer wireless communication hardware, test instruments, diagnostic software, and other miscellaneous lab electronics. In many cases the acquired data are returned in tabular form as pandas data frames.

Name	Contact Info
Dan Kuester (maintainer)	daniel.kuester@nist.gov
Paul Blanchard	formerly with NIST
Alex Curtin	formerly with NIST
Keith Forsyth	keith.forsyth@nist.gov
Ryan Jacobs	formerly with NIST
John Ladbury	john.ladbury@nist.gov
Yao Ma	yao.ma@nist.gov
Duncan McGillivray	duncan.a.mcgillivray@nist.gov
Audrey Puls	formerly with NIST
Andre Rosete	formerly with NIST
Michael Voecks	michael.voecks@nist.gov

CHAPTER

ONE

GETTING STARTED WITH `SSMDEVICES`

1.1 Installation

1. Ensure python 3.8 or newer is installed
2. In a command prompt environment for this python interpreter, run `pip install git+https://github.com/usnistgov/ssmdevices`
3. If you need support for VISA instruments, install an NI VISA runtime, for example from here.

Note: Certain commercial equipment, instruments, and software are identified here in order to help specify experimental procedures. Such identification is not intended to imply recommendation or endorsement of any product or service by NIST, nor is it intended to imply that the materials or equipment identified are necessarily the best available for the purpose.

1.2 Documentation

- API Reference
- Examples

1.3 See also

- labbench the base library to develop these device wrappers

CHAPTER

TWO

LICENSING

2.1 NIST License

2.2 Bundled software

The following listing is good-faith understanding of the licensing information of bundled libraries and source code, which are all separately available as open source software. It is provided as a convenience, but should be verified by checking with the owner.

2.2.1 Changed

The following are included as part of this source distribution with modifications.

- A modified version of pyminicircuits is included in *minicircuits.py*: [MIT license](https://github.com/pyvisa/pyvisa/blob/master/LICENSE).

2.2.2 Unchanged

The following are included unchanged as part of this source distribution.

- Windows library binaries `adb.exe`, `AdbWinApi.dll`, ``AdbWinUsbApi.dll``for adb: Apache 2.0 license
- windows executables `IPerf.exe` and `IPerf3.exe` and android executable `iperf`: BSD license
- cygwin1.dll: LGPL version 3

CHAPTER

THREE

SSMDEVICES API

The ssmdevices API organized as a collection of independent device wrappers for different instruments. The wrapper for each specific hardware model is encapsulated within its own class. As such, in many cases, it is possible to copy and adjust source code file that defines that class from the `` `ssmdevices `` repository <https://github.com/usnistgov/ssmdevices/tree/main/ssmdevices>`_. If you implement a variant of the code to operate in your experiments, please feel free to open an issue to share your code so that we can fold your device back into the code base!

The wrapper objects here are implemented on labbench. An understanding of that module is not necessary to use these objects. However, labbench includes many useful tools for organizing the operation of multiple devices. Leveraging those capabilities can help to produce concise code that reads like pseudocode for an experimental procedure.

3.1 ssmdevices.electronics package

`class ssmdevices.electronics.`**`AcronameUSBHub2x4`**`(`*resource: str = None*`)`

Bases: `Device`

A USB hub with control over each port.

`close()`

Release control over the device.

`concurrency`

`data0_enabled`

`data1_enabled`

`data2_enabled`

`data3_enabled`

`enable``(`*data=True*, *power=True*, *channel='all'*`)`

Enable or disable of USB port features at one or all hub ports.

Parameters

- **`data`** – Enables data on the port (if evaluates to true)
- **`power`** – Enables power on the port (if evaluates to true)
- **`channel`** – An integer port number specifies the port to act on, otherwise 'all' (the default) applies the port settings to all ports on the hub.

`isopen`

model = 17

open()

Backend implementations overload this to open a backend connection to the resource.

power0_enabled

power1_enabled

power2_enabled

power3_enabled

resource

set_key(*key*, *value*, *name=None*)

Apply an instrument setting to the instrument. The value ``value" will be applied to the trait attriute ``attr" in type(self).

class ssmdevices.electronics.**SwiftNavPiksi**(*resource: str = ''*, ***, *timeout: float = 2*, *write_termination: bytes = b'\n'*, *baud_rate: int = 1000000*, *parity: bytes = b'N'*, *stopbits: float = 1*, *xonxoff: bool = False*, *rtscts: bool = False*, *dsrdtr: bool = False*, *poll_rate: float = 0.1*, *data_format: bytes = b''*, *stop_timeout: float = 0.5*, *max_queue_size: int = 100000*)

Bases: SerialLoggingDevice

baud_rate: int

concurrency

data_format

dsrdtr

isopen

max_queue_size

parity

poll_rate

resource

rtscts

stop_timeout

stopbits

timeout

write_termination

xonxoff

3.2 ssmdevices.instruments package

`class ssmdevices.instruments.`**`AeroflexTM500`**`(`*resource: str = '127.0.0.1:23', *, timeout: float = 1, ack_timeout: float = 30, busy_retries: int = 20, remote_ip: str = '10.133.0.203', remote_ports: str = '5001 5002 5003', min_acquisition_time: int = 30, port: int = 5003, config_root: str = '.', data_root: str = '.', convert_files: list = []*`)`

Bases: `TelnetDevice`

Control an Aeroflex TM500 network tester with a telnet connection.

The approach here is to iterate through lines of bytes, and add delays as needed for special cases as defined in the *delays* attribute.

At some point, these lines should just be loaded directly from a file that could be treated as a config file.

`ack_timeout`

`arm``(`*scenario_name*`)`

Load the scenario from the command listing in a local TM500 configuration file. The the full path to the configuration file is *os.path.join(self.config_root, self.config_file)+'.conf'* (on the host computer running this python instance).

If the last script that was run is the same as the selected config script, then the script is loaded and sent to the TM500 only if force=True. It always runs on the first call after AeroflexTM500 is instantiated.

Returns
A list of responses to each command sent

`busy_retries`

`close``()`

Disconnect the telnet connection

`static` **`command_log_to_script`**`(`*path*`)`

Scrape a script out of a TM500 “screen save” text file. The output for an input that takes the form <path>/<to>/<filename>.txt will be <path>/<to>/<filename>-script.txt.

`concurrency`

`config_root`

`convert_files`

`data_root`

`isopen`

`min_acquisition_time`

`open``()`

Open a telnet connection to the host defined by the string in self.resource

`port`

`reboot``(`*timeout=180*`)`

Reboot the TMA and TM500 hardware.

remote_ip

remote_ports

resource

stop(*convert=True*)

Stop logging. :param bool convert: Whether to convert the output binary files to text

Returns

If convert=True, a dictionary of {'name': path} items pointing to the converted text output

timeout

trigger()

Start logging and return the path to the directory where the data is being saved.

class ssmdevices.instruments.**ETSLindgrenAzi2005**(*resource: str = ''*, ***, *read_termination: str = '\n'*, *write_termination: str = '\r'*, *timeout: float = 20*, *baud_rate: int = 9600*, *parity: bytes = b'N'*, *stopbits: float = 1*, *xonxoff: bool = False*, *rtscts: bool = False*, *dsrdtr: bool = False*)

Bases: `VISADevice`

baud_rate

cclimit

concurrency

config(*mode*)

cwlimit

define_position

dsrdtr

identity

isopen

options

parity

position

read_termination

resource

rtscts

seek(*value*)

`set_key`(*key*, *value*, *trait_name=None*)

writes an SCPI message to set a parameter with a name key to *value*.

The command message string is formatted as f'{scpi_key} {value}'. This This is automatically called on assignment to property traits that are defined with 'key='.

Parameters

- **scpi_key** (*str*) – the name of the parameter to set
- **value** (*str*) – value to assign
- **name** (*str, None*) – name of the trait setting the key (or None to indicate no trait) (ignored)

`set_limits`(*side*, *value*)

Probably should put some error checking in here to make sure value is a float Also, note we use write here becuase property.setter inserts a space

`set_position`(*value*)

`set_speed`(*value*)

`speed`

`status_byte`

`stop`()

`stopbits`

`timeout`

`whereami`()

`wheredoigo`()

`write_termination`

`xonxoff`

class ssmdevices.instruments.`KeysightU2000XSeries`(*resource: str = ''*, ***, *read_termination: str = '\n'*, *write_termination: str = '\n'*)

Bases: `VISADevice`

Coaxial power sensors connected by USB

`TRIGGER_SOURCES = ('IMM', 'INT', 'EXT', 'BUS', 'INT1')`

`auto_calibration`

`calibrate`() → None

`concurrency`

`fetch`() → float | Series

return power readings from the instrument.

Returns

a single number if trigger_count == 1, otherwise or pandas.Series

`frequency`

`identity`

`initiate_continuous`

`isopen`

`measurement_rate`

`options`

`output_trigger`

`preset`(*wait=True*) → None

restore the instrument to its default state

`read_termination`

`resource`

`status_byte`

`sweep_aperture`

`trigger_count`

`trigger_source`

`write_termination`

class `ssmdevices.instruments.`**`MiniCircuitsRCDAT`**(*resource: str = None*, ***, *usb_path: bytes = None*, *timeout: float = 1*, *frequency: float = None*, *output_power_offset: float = None*, *calibration_path: str = None*, *channel: int = None*)

Bases: `SwitchAttenuatorBase`

`attenuation`

`attenuation_setting`

`calibration_path`

`channel`

`concurrency`

`frequency`

`isopen`

`model`

`output_power`

`output_power_offset`

`resource`

`serial_number`

`timeout`

usb_path

class ssmdevices.instruments.**MiniCircuitsUSBSwitch**(*resource: str = None*, ***, *usb_path: bytes = None*, *timeout: float = 1*)

Bases: `SwitchAttenuatorBase`

concurrency

isopen

model

port

resource

serial_number

timeout

usb_path

class ssmdevices.instruments.**RigolDP800Series**(*resource: str = ''*, ***, *read_termination: str = '\n'*, *write_termination: str = '\n'*)

Bases: `VISADevice`

REMAP_BOOL = {False: 'OFF', True: 'ON'}

concurrency

current1

current2

current3

enable1

enable2

enable3

get_key(*scpi_key*, *trait_name=None*)

This instrument expects keys to have syntax ":COMMAND? PARAM", instead of ":COMMAND PARAM?" as implemented in lb.VISADevice.

Insert the "?" in the appropriate place here.

identity

isopen

open()

Poll *IDN until the instrument responds. Sometimes it needs an extra poke before it responds.

options

read_termination

resource

`set_key`(*scpi_key*, *value*, *trait_name=None*)

This instrument expects sets to have syntax :COMMAND? PARAM,VALUE instead of :COMMAND PARAM VALUE? as implemented in lb.VISADevice.

Implement this behavior here.

`status_byte`

`voltage1`

`voltage2`

`voltage3`

`voltage_setting1`

`voltage_setting2`

`voltage_setting3`

`write_termination`

`class ssmdevices.instruments.RigolOscilloscope`(*resource: str = ''*, ***, *read_termination: str = '\n'*, *write_termination: str = '\n'*)

Bases: `VISADevice`

`concurrency`

`fetch()`

`fetch_rms()`

`identity`

`isopen`

`open`(*horizontal=False*)

opens the instrument.

When managing device connection through a *with* context, this is called automatically and does not need to be invoked.

`options`

`read_termination`

`resource`

`status_byte`

`time_offset`

`time_scale`

`write_termination`

`class ssmdevices.instruments.RohdeSchwarzFSW26Base`(*resource: str = ''*, ***, *read_termination: str = '\n'*, *write_termination: str = '\n'*, *default_window: str = ''*, *default_trace: str = ''*)

Bases: `RohdeSchwarzFSWBase`

amplitude_offset

amplitude_offset_trace2

amplitude_offset_trace3

amplitude_offset_trace4

amplitude_offset_trace5

amplitude_offset_trace6

channel_type

concurrency

default_trace

default_window

display_update

expected_channel_type

format

frequency_center

frequency_span

frequency_start

frequency_stop

identity

initiate_continuous

input_attenuation

input_attenuation_auto

input_preamplifier_enabled

isopen

options

output_trigger2_direction

output_trigger2_type

output_trigger3_direction

output_trigger3_type

read_termination

reference_level

reference_level_trace2

`reference_level_trace3`

`reference_level_trace4`

`reference_level_trace5`

`reference_level_trace6`

`resolution_bandwidth`

`resource`

`status_byte`

`sweep_points`

`sweep_time`

`sweep_time_window2`

`write_termination`

class `ssmdevices.instruments.`**`RohdeSchwarzFSW26IQAnalyzer`**(*resource: str = ''*, ***, *read_termination: str = '\n'*, *write_termination: str = '\n'*, *default_window: str = ''*, *default_trace: str = ''*)

Bases: *RohdeSchwarzFSW26Base*, `RohdeSchwarzIQAnalyzerMixIn`

`amplitude_offset`

`amplitude_offset_trace2`

`amplitude_offset_trace3`

`amplitude_offset_trace4`

`amplitude_offset_trace5`

`amplitude_offset_trace6`

`channel_type`

`concurrency`

`default_trace`

`default_window`

`display_update`

`expected_channel_type`

`format`

`frequency_center`

`frequency_span`

`frequency_start`

`frequency_stop`

`identity`

`initiate_continuous`

`input_attenuation`

`input_attenuation_auto`

`input_preamplifier_enabled`

`isopen`

`options`

`output_trigger2_direction`

`output_trigger2_type`

`output_trigger3_direction`

`output_trigger3_type`

`read_termination`

`reference_level`

`reference_level_trace2`

`reference_level_trace3`

`reference_level_trace4`

`reference_level_trace5`

`reference_level_trace6`

`resolution_bandwidth`

`resource`

`status_byte`

`sweep_points`

`sweep_time`

`sweep_time_window2`

`write_termination`

class `ssmdevices.instruments.`**`RohdeSchwarzFSW26LTEAnalyzer`**(*resource: str = '', *, read_termination: str = '\n', write_termination: str = '\n', default_window: str = '', default_trace: str = ''*)

Bases: *RohdeSchwarzFSW26Base*, `RohdeSchwarzLTEAnalyzerMixIn`

`amplitude_offset`

amplitude_offset_trace2

amplitude_offset_trace3

amplitude_offset_trace4

amplitude_offset_trace5

amplitude_offset_trace6

channel_type

concurrency

default_trace

default_window

display_update

expected_channel_type

format

frequency_center

frequency_span

frequency_start

frequency_stop

identity

initiate_continuous

input_attenuation

input_attenuation_auto

input_preamplifier_enabled

isopen

options

output_trigger2_direction

output_trigger2_type

output_trigger3_direction

output_trigger3_type

read_termination

reference_level

reference_level_trace2

reference_level_trace3

reference_level_trace4

reference_level_trace5

reference_level_trace6

resolution_bandwidth

resource

status_byte

sweep_points

sweep_time

sweep_time_window2

write_termination

class ssmdevices.instruments.**RohdeSchwarzFSW26RealTime**(*resource: str = ''*, ***, *read_termination: str = '\n'*, *write_termination: str = '\n'*, *default_window: str = ''*, *default_trace: str = ''*)

Bases: *RohdeSchwarzFSW26Base*, RohdeSchwarzRealTimeMixIn

amplitude_offset

amplitude_offset_trace2

amplitude_offset_trace3

amplitude_offset_trace4

amplitude_offset_trace5

amplitude_offset_trace6

channel_type

concurrency

default_trace

default_window

display_update

expected_channel_type

format

frequency_center

frequency_span

frequency_start

frequency_stop

identity

initiate_continuous

input_attenuation

input_attenuation_auto

input_preamplifier_enabled

isopen

options

output_trigger2_direction

output_trigger2_type

output_trigger3_direction

output_trigger3_type

read_termination

reference_level

reference_level_trace2

reference_level_trace3

reference_level_trace4

reference_level_trace5

reference_level_trace6

resolution_bandwidth

resource

status_byte

sweep_points

sweep_time

sweep_time_window2

write_termination

class ssmdevices.instruments.**RohdeSchwarzFSW26SpectrumAnalyzer**(*resource: str = '', *, read_termination: str = '\n', write_termination: str = '\n', default_window: str = '', default_trace: str = ''*)

Bases: *RohdeSchwarzFSW26Base*, RohdeSchwarzSpectrumAnalyzerMixIn

amplitude_offset

amplitude_offset_trace2

amplitude_offset_trace3

amplitude_offset_trace4

amplitude_offset_trace5

amplitude_offset_trace6

channel_type

concurrency

default_trace

default_window

display_update

expected_channel_type

format

frequency_center

frequency_span

frequency_start

frequency_stop

identity

initiate_continuous

input_attenuation

input_attenuation_auto

input_preamplifier_enabled

isopen

options

output_trigger2_direction

output_trigger2_type

output_trigger3_direction

output_trigger3_type

read_termination

reference_level

reference_level_trace2

reference_level_trace3

reference_level_trace4

`reference_level_trace5`

`reference_level_trace6`

`resolution_bandwidth`

`resource`

`status_byte`

`sweep_points`

`sweep_time`

`sweep_time_window2`

`write_termination`

class `ssmdevices.instruments.`**`RohdeSchwarzFSW43Base`**(*resource: str = ''*, ***, *read_termination: str = '\n'*, *write_termination: str = '\n'*, *default_window: str = ''*, *default_trace: str = ''*)

Bases: `RohdeSchwarzFSWBase`

`amplitude_offset`

`amplitude_offset_trace2`

`amplitude_offset_trace3`

`amplitude_offset_trace4`

`amplitude_offset_trace5`

`amplitude_offset_trace6`

`channel_type`

`concurrency`

`default_trace`

`default_window`

`display_update`

`expected_channel_type`

`format`

`frequency_center`

`frequency_span`

`frequency_start`

`frequency_stop`

`identity`

`initiate_continuous`

input_attenuation

input_attenuation_auto

input_preamplifier_enabled

isopen

options

output_trigger2_direction

output_trigger2_type

output_trigger3_direction

output_trigger3_type

read_termination

reference_level

reference_level_trace2

reference_level_trace3

reference_level_trace4

reference_level_trace5

reference_level_trace6

resolution_bandwidth

resource

status_byte

sweep_points

sweep_time

sweep_time_window2

write_termination

class ssmdevices.instruments.**RohdeSchwarzFSW43IQAnalyzer**(*resource: str = ''*, ***, *read_termination: str = '\n'*, *write_termination: str = '\n'*, *default_window: str = ''*, *default_trace: str = ''*)

Bases: *RohdeSchwarzFSW43Base*, RohdeSchwarzIQAnalyzerMixIn

amplitude_offset

amplitude_offset_trace2

amplitude_offset_trace3

amplitude_offset_trace4

`amplitude_offset_trace5`

`amplitude_offset_trace6`

`channel_type`

`concurrency`

`default_trace`

`default_window`

`display_update`

`expected_channel_type`

`format`

`frequency_center`

`frequency_span`

`frequency_start`

`frequency_stop`

`identity`

`initiate_continuous`

`input_attenuation`

`input_attenuation_auto`

`input_preamplifier_enabled`

`isopen`

`options`

`output_trigger2_direction`

`output_trigger2_type`

`output_trigger3_direction`

`output_trigger3_type`

`read_termination`

`reference_level`

`reference_level_trace2`

`reference_level_trace3`

`reference_level_trace4`

`reference_level_trace5`

`reference_level_trace6`

resolution_bandwidth

resource

status_byte

sweep_points

sweep_time

sweep_time_window2

write_termination

class ssmdevices.instruments.**RohdeSchwarzFSW43LTEAnalyzer**(*resource: str = ''*, ***, *read_termination: str = '\n'*, *write_termination: str = '\n'*, *default_window: str = ''*, *default_trace: str = ''*)

Bases: *RohdeSchwarzFSW43Base*, RohdeSchwarzLTEAnalyzerMixIn

amplitude_offset

amplitude_offset_trace2

amplitude_offset_trace3

amplitude_offset_trace4

amplitude_offset_trace5

amplitude_offset_trace6

channel_type

concurrency

default_trace

default_window

display_update

expected_channel_type

format

frequency_center

frequency_span

frequency_start

frequency_stop

identity

initiate_continuous

input_attenuation

input_attenuation_auto

input_preamplifier_enabled

isopen

options

output_trigger2_direction

output_trigger2_type

output_trigger3_direction

output_trigger3_type

read_termination

reference_level

reference_level_trace2

reference_level_trace3

reference_level_trace4

reference_level_trace5

reference_level_trace6

resolution_bandwidth

resource

status_byte

sweep_points

sweep_time

sweep_time_window2

write_termination

class ssmdevices.instruments.**RohdeSchwarzFSW43RealTime**(*resource: str = ''*, ***, *read_termination: str = '\n'*, *write_termination: str = '\n'*, *default_window: str = ''*, *default_trace: str = ''*)

Bases: *RohdeSchwarzFSW43Base*, RohdeSchwarzRealTimeMixIn

amplitude_offset

amplitude_offset_trace2

amplitude_offset_trace3

amplitude_offset_trace4

amplitude_offset_trace5

`amplitude_offset_trace6`

`channel_type`

`concurrency`

`default_trace`

`default_window`

`display_update`

`expected_channel_type`

`format`

`frequency_center`

`frequency_span`

`frequency_start`

`frequency_stop`

`identity`

`initiate_continuous`

`input_attenuation`

`input_attenuation_auto`

`input_preamplifier_enabled`

`isopen`

`options`

`output_trigger2_direction`

`output_trigger2_type`

`output_trigger3_direction`

`output_trigger3_type`

`read_termination`

`reference_level`

`reference_level_trace2`

`reference_level_trace3`

`reference_level_trace4`

`reference_level_trace5`

`reference_level_trace6`

`resolution_bandwidth`

resource

status_byte

sweep_points

sweep_time

sweep_time_window2

write_termination

class ssmdevices.instruments.**RohdeSchwarzFSW43SpectrumAnalyzer**(*resource: str = ''*, *, *read_termination: str = '\n'*, *write_termination: str = '\n'*, *default_window: str = ''*, *default_trace: str = ''*)

Bases: *RohdeSchwarzFSW43Base*, RohdeSchwarzSpectrumAnalyzerMixIn

amplitude_offset

amplitude_offset_trace2

amplitude_offset_trace3

amplitude_offset_trace4

amplitude_offset_trace5

amplitude_offset_trace6

channel_type

concurrency

default_trace

default_window

display_update

expected_channel_type

format

frequency_center

frequency_span

frequency_start

frequency_stop

identity

initiate_continuous

input_attenuation

input_attenuation_auto

`input_preamplifier_enabled`

`isopen`

`options`

`output_trigger2_direction`

`output_trigger2_type`

`output_trigger3_direction`

`output_trigger3_type`

`read_termination`

`reference_level`

`reference_level_trace2`

`reference_level_trace3`

`reference_level_trace4`

`reference_level_trace5`

`reference_level_trace6`

`resolution_bandwidth`

`resource`

`status_byte`

`sweep_points`

`sweep_time`

`sweep_time_window2`

`write_termination`

class ssmdevices.instruments.**RohdeSchwarzNRP18s**(*resource: str = ''*, ***, *write_termination: str = '\n'*)

Bases: *RohdeSchwarzNRPSeries*

`average_auto`

`average_count`

`average_enable`

`concurrency`

`frequency`

`function`

`identity`

`initiate_continuous`

`isopen`

`options`

`read_termination`

`resource`

`smoothing_enable`

`status_byte`

`trace_average_count`

`trace_average_enable`

`trace_average_mode`

`trace_offset_time`

`trace_points`

`trace_realtime`

`trace_time`

`trigger_count`

`trigger_delay`

`trigger_holdoff`

`trigger_level`

`trigger_source`

`write_termination`

class `ssmdevices.instruments.`**`RohdeSchwarzNRP8s`**(*resource: str = ''*, ***, *write_termination: str = '\n'*)

Bases: *RohdeSchwarzNRPSeries*

`average_auto`

`average_count`

`average_enable`

`concurrency`

`frequency`

`function`

`identity`

`initiate_continuous`

`isopen`

`options`

`read_termination`

`resource`

`smoothing_enable`

`status_byte`

`trace_average_count`

`trace_average_enable`

`trace_average_mode`

`trace_offset_time`

`trace_points`

`trace_realtime`

`trace_time`

`trigger_count`

`trigger_delay`

`trigger_holdoff`

`trigger_level`

`trigger_source`

`write_termination`

class `ssmdevices.instruments.`**`RohdeSchwarzNRPSeries`**(*resource: str = ''*, ***, *write_termination: str = '\n'*)

Bases: `VISADevice`

Coaxial power sensors connected by USB.

These require the installation of proprietary drivers from the vendor website. Resource strings for connections take the form 'RSNRP::0x00e2::103892::INSTR'.

`FUNCTIONS = ('POW:AVG', 'POW:BURS:AVG', 'POW:TSL:AVG', 'XTIM:POW', 'XTIM:POWer')`

`TRIGGER_SOURCES = ('HOLD', 'IMM', 'INT', 'EXT', 'EXT1', 'EXT2', 'BUS', 'INT1')`

`average_auto`

`average_count`

`average_enable`

`concurrency`

`fetch()`

Return a single number or pandas Series containing the power readings

`fetch_buffer()`

Return a single number or pandas Series containing the power readings

`frequency`

`function`

`identity`

`initiate_continuous`

`isopen`

`options`

`preset``()`

sends '*RST' to reset the instrument to preset

`read_termination`

`resource`

`setup_trace``(`*frequency*, *trace_points*, *sample_period*, *trigger_level*, *trigger_delay*, *trigger_source*`)`

Parameters

- **`frequency`** – in Hz
- **`trace_points`** – number of points in the trace (perhaps as high as 5000)
- **`sample_period`** – in s
- **`trigger_level`** – in dBm
- **`trigger_delay`** – in s
- **`trigger_source`** – 'HOLD: No trigger; IMM: Software; INT: Internal level trigger; EXT2: External trigger, 10 kOhm'

Returns

None

`smoothing_enable`

`status_byte`

`trace_average_count`

`trace_average_enable`

`trace_average_mode`

`trace_offset_time`

`trace_points`

`trace_realtime`

`trace_time`

`trigger_count`

`trigger_delay`

`trigger_holdoff`

`trigger_level`

`trigger_single()`

`trigger_source`

`write_termination`

class ssmdevices.instruments.**RohdeSchwarzSMW200A**(*resource: str = ''*, ***, *read_termination: str = '\n'*, *write_termination: str = '\n'*)

Bases: `VISADevice`

`concurrency`

`frequency_center`

`identity`

`isopen`

`load_state`(*FileName*, *opc=False*, *num='4'*)

Loads a previously saved state file in the instrument

Parameters

- **FileName** (*string*) – state file location on the instrument
- **opc** (*bool*) – set the VISA op complete flag?
- **num** (*int*) – state number in the saved filename

`options`

`read_termination`

`resource`

`rf_output_enable`

`rf_output_power`

`save_state`(*FileName*, *num='4'*)

Save current state of the device to the default directory. :param FileName: state file location on the instrument :type FileName: string

Parameters

num (*int*) – state number in the saved filename

`status_byte`

`write_termination`

class ssmdevices.instruments.**RohdeSchwarzZMBSeries**(*resource: str = ''*, ***, *read_termination: str = '\n'*, *write_termination: str = '\n'*)

Bases: `VISADevice`

A network analyzer.

Author: Audrey Puls

`clear()`

`concurrency`

`identity`

`initiate_continuous`

`isopen`

`options`

`read_termination`

`resource`

`save_trace_to_csv`(*path*, *trace=1*)

Save the specified trace to a csv file on the instrument. Block until the operation is finished.

`status_byte`

`trigger`()

Initiate a software trigger.

Consider setting *state.initiate_continuous = False* first so that the instrument waits for this trigger before starting a sweep.

`write_termination`

`class ssmdevices.instruments.SpirentGSS8000`(*resource: str = 'COM17'*, ***, *timeout: float = 2*, *write_termination: bytes = b'\n'*, *baud_rate: int = 9600*, *parity: bytes = b'N'*, *stopbits: float = 1*, *xonxoff: bool = False*, *rtscts: bool = False*, *dsrdtr: bool = False*)

Bases: `SerialDevice`

Control a Spirent GPS GSS8000 simulator over a serial connection.

Responses from the Spirent seem to be incompatible with pyvisa, so this driver uses plain serial.

`abort`()

Force stop the current scenario.

`baud_rate: int`

`concurrency`

`dsrdtr`

`end`()

Stop running the current scenario. If a scenario is not running, an exception is raised.

`static fix_path_name`(*path*)

`get_key`(*key*, *trait_name=None*)

implement this in subclasses to use *key* to retreive a parameter value from the Device with self.backend.

property traits defined with “key=” call this to retrieve values from the backend.

`isopen`

`load_scenario`(*path*)

Load a GPS scenario from a file stored on the instrument.

Parameters

`path` – Full path to scenario file on the instrument.

parity

query(*command*)

reset()

End any currently running scenario, then rewind

resource

rewind()

Rewind the current scenario to the beginning.

rtscts

run()

Start running the current scenario. Requires that there is time left in the scenario, otherwise run *rewind()* first.

running

save_scenario(*folderpath*)

Save the current GPS scenario to a file stored on the instrument.

Parameters
path – Full path to scenario file on the instrument.

status

stopbits

timeout

utc_time

write(*key*, *returns=None*)

Send a message to the spirent, and check the status message returned by the spirent.

Returns
Either 'value' (return the data response), 'status' (return the instrument status), or None (raise an exception if a data value is returned)

write_termination

xonxoff

3.3 ssmdevices.software package

class ssmdevices.software.**IPerf2**(*resource: str = None*, ***, *binary_path: Path = 'C:\\\\Users\\\\dkuester\\\\Documents\\\\src\\\\ssmdevices\\\\ssmdevices\\\\lib\\\\iperf.exe'*, *timeout: float = 5*, *server: bool = False*, *port: int = 5201*, *bind: str = None*, *format: str = None*, *time: float = None*, *number: int = None*, *interval: float = None*, *udp: bool = False*, *bit_rate: str = None*, *buffer_size: int = None*, *tcp_window_size: int = None*, *nodelay: bool = False*, *mss: int = None*, *bidirectional: bool = False*, *report_style: str = 'C'*)

Bases: `_IPerfBase`

Run an instance of iperf to profile data transfer speed. It can operate as a server (listener) or client (sender), operating either in the foreground or as a background thread. When running as an iperf client (server=False).

`DATAFRAME_COLUMNS = ('jitter_milliseconds', 'datagrams_lost', 'datagrams_sent', 'datagrams_loss_percentage', 'datagrams_out_of_order')`

`FLAGS = {'bidirectional': '-d', 'bind': '-B', 'bit_rate': '-b', 'buffer_size': '-l', 'interval': '-i', 'mss': '-M', 'nodelay': '-N', 'number': '-n', 'port': '-p', 'report_style': '-y', 'resource': '-c', 'server': '-s', 'tcp_window_size': '-w', 'time': '-t', 'udp': '-u'}`

`bidirectional`

`binary_path`

`bind`

`bit_rate`

`buffer_size`

`concurrency`

`format`

`interval`

`isopen`

`mss`

`nodelay`

`number`

`port`

`profile`(*block=True*)

`read_stdout()`

retreive text from standard output, and parse into a pandas DataFrame if self.report_style is None

`report_style`

`resource`

`server`

`tcp_window_size`

`time`

`timeout`

`udp`

class ssmdevices.software.**IPerf2BoundPair**(*resource: str = '', *, binary_path: Path = 'C:\\\\Users\\\\dkuester\\\\Documents\\\\src\\\\ssmdevices\\\\ssmdevices\\\\lib\\\\iperf.exe', timeout: float = 5, server: str = '', port: int = 5201, bind: str = None, format: str = None, time: float = None, number: int = None, interval: float = None, udp: bool = False, bit_rate: str = None, buffer_size: int = None, tcp_window_size: int = None, nodelay: bool = False, mss: int = None, bidirectional: bool = False, report_style: str = 'C', client: str = ''*)

Bases: `IPerf2`

Configure and run an iperf client and a server pair on the host.

Outputs from to interfaces in order to ensure that data is routed between them, not through localhost or any other interface.

bidirectional

binary_path

bind

bit_rate

buffer_size

children = {}

client

close()

Backend implementations must overload this to disconnect an existing connection to the resource encapsulated in the object.

concurrency

format

interval

isopen

kill()

If a process is running in the background, kill it. Sends a console warning if no process is running.

mss

nodelay

number

open()

The `open()` method implements opening in the `Device` object protocol. Call the `execute()` method when open to execute the binary.

port

profile(*block=True, **kws*)

read_stdout(*client_ret=None*)

retreive text from standard output, and parse into a pandas DataFrame if self.report_style is None

report_style

resource

running()

Check whether a background process is running.

Returns

True if running, otherwise False

server

tcp_window_size

time

timeout

udp

class ssmdevices.software.**IPerf2OnAndroid**(*resource: str = None*, ***, *binary_path: Path = 'C:\\\\Users\\\\dkuester\\\\Documents\\\\src\\\\ssmdevices\\\\ssmdevices\\\\lib\\\\adb.exe'*, *timeout: float = 5*, *server: bool = False*, *port: int = 5201*, *bind: str = None*, *format: str = None*, *time: float = None*, *number: int = None*, *interval: float = None*, *udp: bool = False*, *bit_rate: str = None*, *buffer_size: int = None*, *tcp_window_size: int = None*, *nodelay: bool = False*, *mss: int = None*, *bidirectional: bool = False*, *report_style: str = 'C'*, *remote_binary_path: str = '/data/local/tmp/iperf'*)

Bases: *IPerf2*

bidirectional

binary_path

bind

bit_rate

buffer_size

concurrency

format

interval

isopen

kill(*wait_time=3*)

Kill the local process and the iperf process on the UE.

mss

nodelay

number

open()

Open an adb connection to the handset, copy the iperf binary onto the phone, and verify that iperf executes.

`port`

`profile`(*block=True*)

`read_stdout`()

adb seems to forward stderr as stdout. Filter out some undesired resulting status messages.

`reboot`(*block=True*)

Reboot the device.

Parameters
`block` – if truey, block until the device is ready to accept commands.

`remote_binary_path`

`report_style`

`resource`

`server`

`tcp_window_size`

`time`

`timeout`

`udp`

`wait_for_cell_data`(*timeout=60*)

Block until cellular data is available

Parameters
`timeout` – how long to wait for a connection before raising a Timeout error

Returns
None

`wait_for_device`(*timeout=30*)

Block until the device is ready to accept commands

Returns
None

`class ssmdevices.software.IPerf3`(*resource: str = None*, ***, *binary_path: Path = 'C:\\\\Users\\\\dkuester\\\\Documents\\\\src\\\\ssmdevices\\\\ssmdevices\\\\lib\\\\iperf3.exe'*, *timeout: float = 5*, *server: bool = False*, *port: int = 5201*, *bind: str = None*, *format: str = None*, *time: float = None*, *number: int = None*, *interval: float = None*, *udp: bool = False*, *bit_rate: str = None*, *buffer_size: int = None*, *tcp_window_size: int = None*, *nodelay: bool = False*, *mss: int = None*, *reverse: bool = False*, *json: bool = False*, *zerocopy: bool = False*)

Bases: `_IPerfBase`

Run an instance of iperf3, collecting output data in a background thread. When running as an iperf client (server=False), The default value is the path that installs with 64-bit cygwin.

FLAGS = {'bind': '-B', 'bit_rate': '-b', 'buffer_size': '-l', 'interval': '-i', 'json': '-J', 'mss': '-M', 'nodelay': '-N', 'number': '-n', 'port': '-p', 'resource': '-c', 'reverse': '-R', 'server': '-s', 'tcp_window_size': '-w', 'time': '-t', 'udp': '-u', 'zerocopy': '-Z'}

binary_path

bind

bit_rate

buffer_size

concurrency

format

interval

isopen

json

mss

nodelay

number

port

resource

reverse

server

tcp_window_size

time

timeout

udp

zerocopy

class ssmdevices.software.**QXDM**(*resource: int = 0*, ***, *cache_path: str = 'temp'*, *connection_timeout: float = 2*)

Bases: Win32ComDevice

QXDM software wrapper

cache_path

close()

Backend implementations must overload this to disconnect an existing connection to the resource encapsulated in the object.

com_object

concurrency

configure(*config_path*, *min_acquisition_time=None*)

Load the QXDM .dmc configuration file at the specified path, with adjustments that disable special file output modes like autosave, quicksave, and automatic segmenting based on time and file size.

connection_timeout

get_key(*key*, *trait_name=None*)

implement this in subclasses to use *key* to retreive a parameter value from the Device with self.backend.

property traits defined with "key=" call this to retrieve values from the backend.

isopen

open()

Connect to the win32 com object

reconnect()

resource

save(*path=None*, *saveNm=None*)

Stop the run and save the data in a file at the specified path. If path is None, autogenerate with self.cache_path and self.data_filename.

This method is threadsafe.

Returns
The absolute path to the data file

start(*wait=True*)

Start acquisition, optionally waiting to return until new data enters the QXDM item store.

ue_build_id

ue_esn

ue_imei

ue_mode

ue_model_number

version

class ssmdevices.software.**TrafficProfiler_ClosedLoopTCP**(*resource: str = ''*, ***, *server: str = ''*, *client: str = ''*, *receive_side: str = ''*, *port: int = 0*, *timeout: float = 2*, *tcp_nodelay: bool = True*, *sync_each: bool = False*, *delay: float = 0*)

Bases: `TrafficProfiler_ClosedLoop`

CONN_WINERRS = (10051,)

PORT_WINERRS = (10013, 10048)

client

concurrency

`delay`

`isopen`

`mss()`

`mtu()`

`port`

`profile_count`(*buffer_size: int*, *count: int*)

sends *count* buffers of size *buffer_size* bytes and returns profiling information"

Parameters

- **buffer_size** (*int*) – number of bytes to send in each buffer
- **count** (*int*) – the number of buffers to send

Returns

a DataFrame indexed on PC time containing columns 'bits_per_second', 'duration', 'delay', 'queuing_duration'

`profile_duration`(*buffer_size: int*, *duration: float*)

sends buffers of size *buffer_size* bytes until *duration* seconds have elapsed, and returns profiling information"

Parameters

- **buffer_size** (*int*) – number of bytes to send in each buffer
- **duration** (*float*) – the minimum number of seconds to spend profiling

Returns

a DataFrame indexed on PC time containing columns 'bits_per_second', 'duration', 'delay', 'queuing_duration'

`receive_side`

`resource`

`server`

`sync_each`

`tcp_nodelay`

`timeout`

`wait_for_interfaces`(*timeout*)

class ssmdevices.software.**WLANClient**(*resource: str = ''*, ***, *ssid: str = None*, *timeout: float = 10*)

Bases: `Device`

`channel`

`concurrency`

`description`

`interface_connect()`

`interface_disconnect()`

Try to disconnect to the WLAN interface, or raise TimeoutError if there is no connection after the specified timeout.

Parameters
`timeout` (*float*) – timeout to wait before raising TimeoutError

`interface_reconnect()`

Reconnect to the network interface.

Returns
time elapsed to reconnect

`isopen`

`isup`

`classmethod list_available_clients`(*by='interface'*)

`open()`

Backend implementations overload this to open a backend connection to the resource.

`refresh()`

`resource`

`signal`

`ssid`

`state`

`timeout`

`transmit_rate_mbps`

`class ssmdevices.software.WLANInfo`(*resource: str = ''*, ***, *binary_path: Path = 'C:\\Windows\\System32\\netsh.exe'*, *timeout: float = 5*, *only_bssid: bool = False*, *interface: str = None*)

Bases: `ShellBackend`

Parse calls to netsh to get information about WLAN interfaces.

`FLAGS = {'interface': 'interface=', 'only_bssid': 'mode=bssid'}`

`binary_path`

`concurrency`

`get_wlan_interfaces`(*name=None*, *param=None*)

`get_wlan_ssids`(*interface*)

`interface`

`isopen`

`only_bssid`

`resource`

`timeout`

`wait``()`

`ssmdevices.software.`**`find_free_port`**`()`

`ssmdevices.software.`**`get_ipv4_address`**`(`*resource*`)`

Try to look up the IP address of a network interface by its name or MAC (physical) address.

If the interface does not exist, the medium is disconnected, or there is no IP address associated with the interface, raise *ConnectionError*.

`ssmdevices.software.`**`get_ipv4_occupied_ports`**`(`*ip*`)`

`ssmdevices.software.`**`list_network_interfaces`**`(`*by='interface'*`)`

`ssmdevices.software.`**`network_interface_info`**`(`*resource*`)`

Try to look up the IP address of a network interface by its name or MAC (physical) address.

If the interface does not exist, the medium is disconnected, or there is no IP address associated with the interface, raise *ConnectionError*.