API reference for ssmdevices

Release 0.12

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

Jun 08, 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
Duncan McGillivray	duncan.a.mcgillivray@nist.gov
Andre Rosete	andre.rosete@nist.gov
Paul Blanchard	paul.blanchard@nist.gov
Michael Voecks	michael.voecks@nist.gov
Ryan Jacobs	ryan.jacobs@nist.gov
Alex Curtin	alexandra.curtin@nist.gov
Audrey Puls	audrey.puls@nist.gov
John Ladbury	john.ladbury@nist.gov
Yao Ma	yao.ma@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

- ssmdevices API
- 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`

True if the device supports threading

Constraints:
sets=False

Type
bool

`data0_enabled`

bool:

`data1_enabled`

bool:

`data2_enabled`

bool:

`data3_enabled`

bool:

`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`

is the backend ready?

Type
bool

`model = 17`

`open`()

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

`power0_enabled`

bool:

`power1_enabled`

bool:

`power2_enabled`

bool:

`power3_enabled`

bool:

`resource`

None to autodetect, or a serial number string

Constraints:
cache=True, allow_none=True

Type
str

`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`

int:

`concurrency`

True if the device supports threading

Constraints:
sets=False

Type
bool

`data_format`

Data format metadata

Type
bytes

`dsrdtr`

True to enable hardware (DSR/DTR) flow control.

Type
bool

`isopen`

is the backend ready?

Type
bool

`max_queue_size`

bytes to allocate in the data retreival buffer

Type
int

`parity`

Parity in the physical serial connection.

Type
bytes

`poll_rate`

Data retreival rate from the device (in seconds)

Type
float

`resource`

platform-dependent serial port address

Type
str

`rtscts`

True to enable hardware (RTS/CTS) flow control.

Type
bool

`stop_timeout`

delay after *stop* before terminating run thread

Type
float

`stopbits`
Number of stop bits, one of *[1, 1.5, or 2.]*.

Type
float

`timeout`
Max time to wait for a connection before raising TimeoutError.

Type
float

`write_termination`
Termination character to send after a write.

Type
bytes

`xonxoff`
True to enable software flow control.

Type
bool

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`
how long to wait for a command acknowledgment from the TM500 (s)

Type
float

`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`

int:

`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`

True if the device supports threading

Constraints:
sets=False

Type
bool

`config_root`

path to the command scripts directory

Type
str

`convert_files`

text to match in the filename of data output files to convert

Type
list

`data_root`

remote save root directory

Type
str

`isopen`

is the backend ready?

Type
bool

`min_acquisition_time`

minimum time to spend acquiring logs (s)

Type
int

`open()`

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

`port`

int:

`reboot`(*timeout=180*)

Reboot the TMA and TM500 hardware.

`remote_ip`

ip address of TM500 backend

Type
str

`remote_ports`

port of TM500 backend

Type
str

`resource`

server host address

Type
str

`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`

leave the timeout small to allow keyboard interrupts

Type
float

`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`

int:

`cclimit`

cclimit

Constraints:
key='LL'

Type
float

`concurrency`

True if the device supports threading

Constraints:
sets=False

Type
bool

`config`(*mode*)

`cwlimit`

cwlimit

Constraints:
key='UL'

Type
float

`define_position`

rotation (degrees)

Constraints:
key='CP'

Type
float

`dsrdtr`

bool:

`identity`

identity string reported by the instrument

Constraints:
key='*IDN', sets=False, cache=True

Type
str

`isopen`

is the backend ready?

Type
bool

`options`

options reported by the instrument

Constraints:
key='*OPT', sets=False, cache=True

Type
str

`parity`

bytes:

`position`

rotation (degrees)

Constraints:
key='SK', gets=False

Type
float

`read_termination`

str:

`resource`

device address or URI

Constraints:
cache=True, allow_none=True

Type
str

`rtscts`

bool:

`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`

speed

Constraints:
key='S'

Type
int

status_byte

instrument status decoded from '*STB?'

Constraints:
sets=False

Type
dict

stop()

stopbits

float:

timeout

float:

whereami()

wheredoigo()

write_termination

str:

xonxoff

bool:

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')

concurrency

True if the device supports threading

Constraints:
sets=False

Type
bool

fetch()

Return a single number or pandas Series containing the power readings

frequency

input signal center frequency (in Hz)

Constraints:
key='SENS:FREQ'

Type
float

`identity`

identity string reported by the instrument

Constraints:
key='*IDN', sets=False, cache=True

Type
str

`initiate_continuous`

bool:

Constraints:
key='INIT:CONT'

`isopen`

is the backend ready?

Type
bool

`measurement_rate`

str:

Constraints:
key='SENS:MRAT', only=('NORM', 'DOUB', 'FAST'), case=False

`options`

options reported by the instrument

Constraints:
key='*OPT', sets=False, cache=True

Type
str

`output_trigger`

bool:

Constraints:
key='OUTP:TRIG'

`preset()`

sends '*RST' to reset the instrument to preset

`read_termination`

end of line string to expect in query replies

Constraints:
cache=True

Type
str

`resource`

device address or URI

Constraints:
cache=True, allow_none=True

Type
str

`status_byte`

instrument status decoded from '*STB?'

Constraints:
sets=False

Type
dict

`sweep_aperture`

time

Constraints:
key='SWE:APER'

Type
float (s)

`trigger_count`

int:

Constraints:
key='TRIG:COUN'

`trigger_source`

str:

Constraints:
key='TRIG:SOUR', only=('IMM', 'INT', 'EXT', 'BUS', 'INT1'), case=False

`write_termination`

end of line string to send after writes

Constraints:
cache=True

Type
str

`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`

calibrated attenuation

Constraints:
allow_none=False

Type
float (dB)

`attenuation_setting`

uncalibrated attenuation

Type
float (dB)

`calibration_path`

path to the calibration table csv file (containing frequency (row) and attenuation setting (column)), or None to search ssmdevices

Constraints:
cache=True, allow_none=True

Type
str

`channel`

a port selector for 4 port attenuators None is a single attenuator

Constraints:
cache=True

Type
int

`concurrency`

True if the device supports threading

Constraints:
sets=False

Type
bool

`frequency`

frequency for calibration data (None for no calibration)

Type
float (Hz)

`isopen`

is the backend ready?

Type
bool

`model`

str:

Constraints:
sets=False, cache=True

`output_power`

(-1*(calibrated attenuation)) + value.float(label='dBm')

Constraints:
allow_none=False

Type
float (dB)

`output_power_offset`

output power level at 0 dB attenuation

Type
float (dBm)

`resource`

serial number; must be set if more than one device is connected

Constraints:
cache=True, allow_none=True

Type
str

`serial_number`

str:

Constraints:
sets=False, cache=True

`timeout`

float (s):

Constraints:
cache=True

`usb_path`

override *resource* to connect to a specific USB path

Constraints:
cache=True, allow_none=True

Type
bytes

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

Bases: `SwitchAttenuatorBase`

`concurrency`

True if the device supports threading

Constraints:
sets=False

Type
bool

`isopen`

is the backend ready?

Type
bool

`model`

str:

Constraints:
sets=False, cache=True

`port`

int:

`resource`

serial number; must be set if more than one device is connected

Constraints:
cache=True, allow_none=True

Type
str

`serial_number`

str:

Constraints:
sets=False, cache=True

`timeout`

float (s):

Constraints:
cache=True

`usb_path`

override *resource* to connect to a specific USB path

Constraints:
cache=True, allow_none=True

Type
bytes

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

Bases: `VISADevice`

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

`concurrency`

True if the device supports threading

Constraints:
sets=False

Type
bool

`current1`

current draw reading on channel 1

Constraints:
key=':MEAS:CURR CH1', sets=False

Type
float

`current2`

current draw reading on channel 2

Constraints:
key=':MEAS:CURR CH2', sets=False

Type
float

`current3`

current draw reading on channel 3

Constraints:
key=':MEAS:CURR CH3', sets=False

Type
float

`enable1`

enable DC output on channel 1

Constraints:
key=':OUTP CH1', remap={False: 'OFF', True: 'ON'}

Type
bool

`enable2`

enable DC output on channel 2

Constraints:
key=':OUTP CH2', remap={False: 'OFF', True: 'ON'}

Type
bool

`enable3`

enable DC output on channel 3

Constraints:
key=':OUTP CH3', remap={False: 'OFF', True: 'ON'}

Type
bool

`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`

identity string reported by the instrument

Constraints:
key='*IDN', sets=False, cache=True

Type
str

`isopen`

is the backend ready?

Type
bool

`open`()

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

`options`

options reported by the instrument

Constraints:
key='*OPT', sets=False, cache=True

Type
str

`read_termination`

end of line string to expect in query replies

Constraints:
cache=True

Type
str

`resource`

device address or URI

Constraints:
cache=True, allow_none=True

Type
str

`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`

instrument status decoded from '*STB?'

Constraints:
sets=False

Type
dict

`voltage1`

output voltage reading on channel 1

Constraints:
key=':MEAS:VOLT CH1', sets=False

Type
float

`voltage2`

output voltage reading channel 2

Constraints:
key=':MEAS:VOLT CH2', sets=False

Type
float

`voltage3`

output voltage reading channel 3

Constraints:
key=':MEAS:VOLT CH3', sets=False

Type
float

`voltage_setting1`

output voltage setting on channel 1

Constraints:
key=':SOUR1:VOLT'

Type
float

`voltage_setting2`

output voltage setting on channel 2

Constraints:
key=':SOUR2:VOLT'

Type
float

`voltage_setting3`

output voltage setting on channel 3

Constraints:
key=':SOUR3:VOLT'

Type
float

`write_termination`

end of line string to send after writes

Constraints:
cache=True

Type
str

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

Bases: `VISADevice`

`concurrency`

True if the device supports threading

Constraints:
sets=False

Type
bool

`fetch()`

`fetch_rms()`

`identity`

identity string reported by the instrument

Constraints:
key='*IDN', sets=False, cache=True

Type
str

`isopen`

is the backend ready?

Type
bool

`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`

options reported by the instrument

Constraints:
key='*OPT', sets=False, cache=True

Type
str

`read_termination`

end of line string to expect in query replies

Constraints:
cache=True

Type
str

`resource`

device address or URI

Constraints:
cache=True, allow_none=True

Type
str

`status_byte`

instrument status decoded from '*STB?'

Constraints:
sets=False

Type
dict

`time_offset`

float (s):

Constraints:
key=':TIM:OFFS'

`time_scale`

float (s):

Constraints:
key=':TIM:SCAL'

`write_termination`

end of line string to send after writes

Constraints:
cache=True

Type
str

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

Bases: RohdeSchwarzFSWBase

amplitude_offset

float (dB):

Constraints:
key='DISP:TRAC1:Y:RLEV:OFFS'

amplitude_offset_trace2

float (dB):

Constraints:
key='DISP:TRAC2:Y:RLEV:OFFS'

amplitude_offset_trace3

float (dB):

Constraints:
key='DISP:TRAC3:Y:RLEV:OFFS'

amplitude_offset_trace4

float (dB):

Constraints:
key='DISP:TRAC4:Y:RLEV:OFFS'

amplitude_offset_trace5

float (dB):

Constraints:
key='DISP:TRAC5:Y:RLEV:OFFS'

amplitude_offset_trace6

float (dB):

Constraints:
key='DISP:TRAC6:Y:RLEV:OFFS'

channel_type

str:

Constraints:
key='INST', only=(None, 'SAN', 'IQ', 'RTIM'), case=False

concurrency

True if the device supports threading

Constraints:
sets=False

Type
bool

default_trace

data trace number to use if unspecified

Constraints:
cache=True

Type
str

`default_window`
data window number to use if unspecified

Constraints:
cache=True

Type
str

`display_update`
bool:

Constraints:
key='SYST:DISP:UPD', remap={False: 'OFF', True: 'ON'}

`expected_channel_type`
which channel type to use

Constraints:
sets=False, cache=True, only=(None, 'SAN', 'IQ', 'RTIM'), allow_none=True

Type
str

`format`
str:

Constraints:
key='FORM', only=('ASC,0', 'REAL,32', 'REAL,64', 'REAL,16'), case=False

`frequency_center`
float (Hz):

Constraints:
key='FREQ:CENT'

`frequency_span`
float (Hz):

Constraints:
key='FREQ:SPAN'

`frequency_start`
float (Hz):

Constraints:
key='FREQ:START'

`frequency_stop`
float (Hz):

Constraints:
key='FREQ:STOP'

`identity`

identity string reported by the instrument

Constraints:
key='*IDN', sets=False, cache=True

Type
str

`initiate_continuous`

bool:

Constraints:
key='INIT:CONT', remap={False: '0', True: '1'}

`input_attenuation`

float:

Constraints:
key='INP:ATT'

`input_attenuation_auto`

bool:

Constraints:
key='INP:ATT:AUTO', remap={False: '0', True: '1'}

`input_preamplifier_enabled`

bool:

Constraints:
key='INP:GAIN:STATE', remap={False: '0', True: '1'}

`isopen`

is the backend ready?

Type
bool

`options`

options reported by the instrument

Constraints:
key='*OPT', sets=False, cache=True

Type
str

`output_trigger2_direction`

str:

Constraints:
key='OUTP:TRIG2:DIR', only=('INP', 'OUTP'), case=False

`output_trigger2_type`

str:

Constraints:
key='OUTP:TRIG2:OTYP', only=('DEV', 'TARM', 'UDEF'), case=False

`output_trigger3_direction`

str:

Constraints:
key='OUTP:TRIG3:DIR', only=('INP', 'OUTP'), case=False

`output_trigger3_type`

str:

Constraints:
key='OUTP:TRIG3:OTYP', only=('DEV', 'TARM', 'UDEF'), case=False

`read_termination`

end of line string to expect in query replies

Constraints:
cache=True

Type
str

`reference_level`

float (dB):

Constraints:
key='DISP:TRAC1:Y:RLEV'

`reference_level_trace2`

float (dB):

Constraints:
key='DISP:TRAC2:Y:RLEV'

`reference_level_trace3`

float (dB):

Constraints:
key='DISP:TRAC3:Y:RLEV'

`reference_level_trace4`

float (dB):

Constraints:
key='DISP:TRAC4:Y:RLEV'

`reference_level_trace5`

float (dB):

Constraints:
key='DISP:TRAC5:Y:RLEV'

`reference_level_trace6`

float (dB):

Constraints:
key='DISP:TRAC6:Y:RLEV'

`resolution_bandwidth`

float (Hz):

Constraints:
key='BAND'

`resource`

device address or URI

Constraints:
cache=True, allow_none=True

Type
str

`status_byte`

instrument status decoded from '*STB?'

Constraints:
sets=False

Type
dict

`sweep_points`

int:

Constraints:
key='SWE:POIN'

`sweep_time`

float (Hz):

Constraints:
key='SWE:TIME'

`sweep_time_window2`

float (Hz):

Constraints:
key='SENS2:SWE:TIME'

`write_termination`

end of line string to send after writes

Constraints:
cache=True

Type
str

`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`

float (dB):

Constraints:
key='DISP:TRAC1:Y:RLEV:OFFS'

`amplitude_offset_trace2`

float (dB):

Constraints:
key='DISP:TRAC2:Y:RLEV:OFFS'

`amplitude_offset_trace3`

float (dB):

Constraints:
key='DISP:TRAC3:Y:RLEV:OFFS'

`amplitude_offset_trace4`

float (dB):

Constraints:
key='DISP:TRAC4:Y:RLEV:OFFS'

`amplitude_offset_trace5`

float (dB):

Constraints:
key='DISP:TRAC5:Y:RLEV:OFFS'

`amplitude_offset_trace6`

float (dB):

Constraints:
key='DISP:TRAC6:Y:RLEV:OFFS'

`channel_type`

str:

Constraints:
key='INST', only=(None, 'SAN', 'IQ', 'RTIM'), case=False

`concurrency`

True if the device supports threading

Constraints:
sets=False

Type
bool

`default_trace`

data trace number to use if unspecified

Constraints:
cache=True

Type
str

`default_window`

data window number to use if unspecified

Constraints:
cache=True

Type
str

`display_update`

bool:

Constraints:
key='SYST:DISP:UPD', remap={False: 'OFF', True: 'ON'}

`expected_channel_type`

which channel type to use

Constraints:
sets=False, cache=True, only=(None, 'SAN', 'IQ', 'RTIM'), allow_none=True

Type
str

`format`

str:

Constraints:
key='FORM', only=('ASC,0', 'REAL,32', 'REAL,64', 'REAL,16'), case=False

`frequency_center`

float (Hz):

Constraints:
key='FREQ:CENT'

`frequency_span`

float (Hz):

Constraints:
key='FREQ:SPAN'

`frequency_start`

float (Hz):

Constraints:
key='FREQ:START'

`frequency_stop`

float (Hz):

Constraints:
key='FREQ:STOP'

`identity`

identity string reported by the instrument

Constraints:
key='*IDN', sets=False, cache=True

Type
str

`initiate_continuous`

bool:

Constraints:
key='INIT:CONT', remap={False: '0', True: '1'}

`input_attenuation`

float:

Constraints:
key='INP:ATT'

`input_attenuation_auto`

bool:

Constraints:
key='INP:ATT:AUTO', remap={False: '0', True: '1'}

`input_preamplifier_enabled`

bool:

Constraints:
key='INP:GAIN:STATE', remap={False: '0', True: '1'}

`isopen`

is the backend ready?

Type
bool

`options`

options reported by the instrument

Constraints:
key='*OPT', sets=False, cache=True

Type
str

`output_trigger2_direction`

str:

Constraints:
key='OUTP:TRIG2:DIR', only=('INP', 'OUTP'), case=False

`output_trigger2_type`

str:

Constraints:
key='OUTP:TRIG2:OTYP', only=('DEV', 'TARM', 'UDEF'), case=False

`output_trigger3_direction`

str:

Constraints:
key='OUTP:TRIG3:DIR', only=('INP', 'OUTP'), case=False

`output_trigger3_type`
: str:

  Constraints:
  key='OUTP:TRIG3:OTYP', only=('DEV', 'TARM', 'UDEF'), case=False

`read_termination`
: end of line string to expect in query replies

  Constraints:
  cache=True

  Type
  str

`reference_level`
: float (dB):

  Constraints:
  key='DISP:TRAC1:Y:RLEV'

`reference_level_trace2`
: float (dB):

  Constraints:
  key='DISP:TRAC2:Y:RLEV'

`reference_level_trace3`
: float (dB):

  Constraints:
  key='DISP:TRAC3:Y:RLEV'

`reference_level_trace4`
: float (dB):

  Constraints:
  key='DISP:TRAC4:Y:RLEV'

`reference_level_trace5`
: float (dB):

  Constraints:
  key='DISP:TRAC5:Y:RLEV'

`reference_level_trace6`
: float (dB):

  Constraints:
  key='DISP:TRAC6:Y:RLEV'

`resolution_bandwidth`
: float (Hz):

  Constraints:
  key='BAND'

`resource`
: device address or URI

Constraints:
cache=True, allow_none=True

Type
str

`status_byte`

instrument status decoded from '*STB?'

Constraints:
sets=False

Type
dict

`sweep_points`

int:

Constraints:
key='SWE:POIN'

`sweep_time`

float (Hz):

Constraints:
key='SWE:TIME'

`sweep_time_window2`

float (Hz):

Constraints:
key='SENS2:SWE:TIME'

`write_termination`

end of line string to send after writes

Constraints:
cache=True

Type
str

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`

float (dB):

Constraints:
key='DISP:TRAC1:Y:RLEV:OFFS'

`amplitude_offset_trace2`

float (dB):

Constraints:
key='DISP:TRAC2:Y:RLEV:OFFS'

`amplitude_offset_trace3`

float (dB):

Constraints:
key='DISP:TRAC3:Y:RLEV:OFFS'

`amplitude_offset_trace4`

float (dB):

Constraints:
key='DISP:TRAC4:Y:RLEV:OFFS'

`amplitude_offset_trace5`

float (dB):

Constraints:
key='DISP:TRAC5:Y:RLEV:OFFS'

`amplitude_offset_trace6`

float (dB):

Constraints:
key='DISP:TRAC6:Y:RLEV:OFFS'

`channel_type`

str:

Constraints:
key='INST', only=(None, 'SAN', 'IQ', 'RTIM'), case=False

`concurrency`

True if the device supports threading

Constraints:
sets=False

Type
bool

`default_trace`

data trace number to use if unspecified

Constraints:
cache=True

Type
str

`default_window`

data window number to use if unspecified

Constraints:
cache=True

Type
str

`display_update`
: bool:

  Constraints:
  : key='SYST:DISP:UPD', remap={False: 'OFF', True: 'ON'}

`expected_channel_type`
: which channel type to use

  Constraints:
  : sets=False, cache=True, only=(None, 'SAN', 'IQ', 'RTIM'), allow_none=True

    Type
    : str

`format`
: str:

  Constraints:
  : key='FORM', only=('ASC,0', 'REAL,32', 'REAL,64', 'REAL,16'), case=False

`frequency_center`
: float (Hz):

  Constraints:
  : key='FREQ:CENT'

`frequency_span`
: float (Hz):

  Constraints:
  : key='FREQ:SPAN'

`frequency_start`
: float (Hz):

  Constraints:
  : key='FREQ:START'

`frequency_stop`
: float (Hz):

  Constraints:
  : key='FREQ:STOP'

`identity`
: identity string reported by the instrument

  Constraints:
  : key='*IDN', sets=False, cache=True

    Type
    : str

`initiate_continuous`
: bool:

  Constraints:
  : key='INIT:CONT', remap={False: '0', True: '1'}

`input_attenuation`
float:

Constraints:
key='INP:ATT'

`input_attenuation_auto`
bool:

Constraints:
key='INP:ATT:AUTO', remap={False: '0', True: '1'}

`input_preamplifier_enabled`
bool:

Constraints:
key='INP:GAIN:STATE', remap={False: '0', True: '1'}

`isopen`
is the backend ready?

Type
bool

`options`
options reported by the instrument

Constraints:
key='*OPT', sets=False, cache=True

Type
str

`output_trigger2_direction`
str:

Constraints:
key='OUTP:TRIG2:DIR', only=('INP', 'OUTP'), case=False

`output_trigger2_type`
str:

Constraints:
key='OUTP:TRIG2:OTYP', only=('DEV', 'TARM', 'UDEF'), case=False

`output_trigger3_direction`
str:

Constraints:
key='OUTP:TRIG3:DIR', only=('INP', 'OUTP'), case=False

`output_trigger3_type`
str:

Constraints:
key='OUTP:TRIG3:OTYP', only=('DEV', 'TARM', 'UDEF'), case=False

`read_termination`
end of line string to expect in query replies

Constraints:
cache=True

Type
str

`reference_level`
float (dB):

Constraints:
key='DISP:TRAC1:Y:RLEV'

`reference_level_trace2`
float (dB):

Constraints:
key='DISP:TRAC2:Y:RLEV'

`reference_level_trace3`
float (dB):

Constraints:
key='DISP:TRAC3:Y:RLEV'

`reference_level_trace4`
float (dB):

Constraints:
key='DISP:TRAC4:Y:RLEV'

`reference_level_trace5`
float (dB):

Constraints:
key='DISP:TRAC5:Y:RLEV'

`reference_level_trace6`
float (dB):

Constraints:
key='DISP:TRAC6:Y:RLEV'

`resolution_bandwidth`
float (Hz):

Constraints:
key='BAND'

`resource`
device address or URI

Constraints:
cache=True, allow_none=True

Type
str

`status_byte`

instrument status decoded from '*STB?'

Constraints:
sets=False

Type
dict

`sweep_points`

int:

Constraints:
key='SWE:POIN'

`sweep_time`

float (Hz):

Constraints:
key='SWE:TIME'

`sweep_time_window2`

float (Hz):

Constraints:
key='SENS2:SWE:TIME'

`write_termination`

end of line string to send after writes

Constraints:
cache=True

Type
str

`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`

float (dB):

Constraints:
key='DISP:TRAC1:Y:RLEV:OFFS'

`amplitude_offset_trace2`

float (dB):

Constraints:
key='DISP:TRAC2:Y:RLEV:OFFS'

`amplitude_offset_trace3`

float (dB):

Constraints:
key='DISP:TRAC3:Y:RLEV:OFFS'

`amplitude_offset_trace4`

float (dB):

Constraints:
key='DISP:TRAC4:Y:RLEV:OFFS'

`amplitude_offset_trace5`

float (dB):

Constraints:
key='DISP:TRAC5:Y:RLEV:OFFS'

`amplitude_offset_trace6`

float (dB):

Constraints:
key='DISP:TRAC6:Y:RLEV:OFFS'

`channel_type`

str:

Constraints:
key='INST', only=(None, 'SAN', 'IQ', 'RTIM'), case=False

`concurrency`

True if the device supports threading

Constraints:
sets=False

Type
bool

`default_trace`

data trace number to use if unspecified

Constraints:
cache=True

Type
str

`default_window`

data window number to use if unspecified

Constraints:
cache=True

Type
str

`display_update`

bool:

Constraints:
key='SYST:DISP:UPD', remap={False: 'OFF', True: 'ON'}

`expected_channel_type`

which channel type to use

Constraints:
sets=False, cache=True, only=(None, ‘SAN’, ‘IQ’, ‘RTIM’), allow_none=True

Type
str

`format`

str:

Constraints:
key=’FORM’, only=(‘ASC,0’, ‘REAL,32’, ‘REAL,64’, ‘REAL,16’), case=False

`frequency_center`

float (Hz):

Constraints:
key=’FREQ:CENT’

`frequency_span`

float (Hz):

Constraints:
key=’FREQ:SPAN’

`frequency_start`

float (Hz):

Constraints:
key=’FREQ:START’

`frequency_stop`

float (Hz):

Constraints:
key=’FREQ:STOP’

`identity`

identity string reported by the instrument

Constraints:
key=’*IDN’, sets=False, cache=True

Type
str

`initiate_continuous`

bool:

Constraints:
key=’INIT:CONT’, remap={False: ‘0’, True: ‘1’}

`input_attenuation`

float:

Constraints:
key=’INP:ATT’

`input_attenuation_auto`

bool:

Constraints:
key='INP:ATT:AUTO', remap={False: '0', True: '1'}

`input_preamplifier_enabled`

bool:

Constraints:
key='INP:GAIN:STATE', remap={False: '0', True: '1'}

`isopen`

is the backend ready?

Type
bool

`options`

options reported by the instrument

Constraints:
key='*OPT', sets=False, cache=True

Type
str

`output_trigger2_direction`

str:

Constraints:
key='OUTP:TRIG2:DIR', only=('INP', 'OUTP'), case=False

`output_trigger2_type`

str:

Constraints:
key='OUTP:TRIG2:OTYP', only=('DEV', 'TARM', 'UDEF'), case=False

`output_trigger3_direction`

str:

Constraints:
key='OUTP:TRIG3:DIR', only=('INP', 'OUTP'), case=False

`output_trigger3_type`

str:

Constraints:
key='OUTP:TRIG3:OTYP', only=('DEV', 'TARM', 'UDEF'), case=False

`read_termination`

end of line string to expect in query replies

Constraints:
cache=True

Type
str

`reference_level`

float (dB):

Constraints:
key='DISP:TRAC1:Y:RLEV'

`reference_level_trace2`

float (dB):

Constraints:
key='DISP:TRAC2:Y:RLEV'

`reference_level_trace3`

float (dB):

Constraints:
key='DISP:TRAC3:Y:RLEV'

`reference_level_trace4`

float (dB):

Constraints:
key='DISP:TRAC4:Y:RLEV'

`reference_level_trace5`

float (dB):

Constraints:
key='DISP:TRAC5:Y:RLEV'

`reference_level_trace6`

float (dB):

Constraints:
key='DISP:TRAC6:Y:RLEV'

`resolution_bandwidth`

float (Hz):

Constraints:
key='BAND'

`resource`

device address or URI

Constraints:
cache=True, allow_none=True

Type
str

`status_byte`

instrument status decoded from '*STB?'

Constraints:
sets=False

Type
dict

`sweep_points`

int:

Constraints:
key='SWE:POIN'

`sweep_time`

float (Hz):

Constraints:
key='SWE:TIME'

`sweep_time_window2`

float (Hz):

Constraints:
key='SENS2:SWE:TIME'

`write_termination`

end of line string to send after writes

Constraints:
cache=True

Type
str

`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`

float (dB):

Constraints:
key='DISP:TRAC1:Y:RLEV:OFFS'

`amplitude_offset_trace2`

float (dB):

Constraints:
key='DISP:TRAC2:Y:RLEV:OFFS'

`amplitude_offset_trace3`

float (dB):

Constraints:
key='DISP:TRAC3:Y:RLEV:OFFS'

`amplitude_offset_trace4`

float (dB):

Constraints:
key='DISP:TRAC4:Y:RLEV:OFFS'

`amplitude_offset_trace5`

float (dB):

Constraints:
key='DISP:TRAC5:Y:RLEV:OFFS'

`amplitude_offset_trace6`

float (dB):

Constraints:
key='DISP:TRAC6:Y:RLEV:OFFS'

`channel_type`

str:

Constraints:
key='INST', only=(None, 'SAN', 'IQ', 'RTIM'), case=False

`concurrency`

True if the device supports threading

Constraints:
sets=False

Type
bool

`default_trace`

data trace number to use if unspecified

Constraints:
cache=True

Type
str

`default_window`

data window number to use if unspecified

Constraints:
cache=True

Type
str

`display_update`

bool:

Constraints:
key='SYST:DISP:UPD', remap={False: 'OFF', True: 'ON'}

`expected_channel_type`

which channel type to use

Constraints:
sets=False, cache=True, only=(None, 'SAN', 'IQ', 'RTIM'), allow_none=True

Type
str

`format`
str:

Constraints:
key='FORM', only=('ASC,0', 'REAL,32', 'REAL,64', 'REAL,16'), case=False

`frequency_center`
float (Hz):

Constraints:
key='FREQ:CENT'

`frequency_span`
float (Hz):

Constraints:
key='FREQ:SPAN'

`frequency_start`
float (Hz):

Constraints:
key='FREQ:START'

`frequency_stop`
float (Hz):

Constraints:
key='FREQ:STOP'

`identity`
identity string reported by the instrument

Constraints:
key='*IDN', sets=False, cache=True

Type
str

`initiate_continuous`
bool:

Constraints:
key='INIT:CONT', remap={False: '0', True: '1'}

`input_attenuation`
float:

Constraints:
key='INP:ATT'

`input_attenuation_auto`
bool:

Constraints:
key='INP:ATT:AUTO', remap={False: '0', True: '1'}

`input_preamplifier_enabled`

bool:

Constraints:
key='INP:GAIN:STATE', remap={False: '0', True: '1'}

`isopen`

is the backend ready?

Type
bool

`options`

options reported by the instrument

Constraints:
key='*OPT', sets=False, cache=True

Type
str

`output_trigger2_direction`

str:

Constraints:
key='OUTP:TRIG2:DIR', only=('INP', 'OUTP'), case=False

`output_trigger2_type`

str:

Constraints:
key='OUTP:TRIG2:OTYP', only=('DEV', 'TARM', 'UDEF'), case=False

`output_trigger3_direction`

str:

Constraints:
key='OUTP:TRIG3:DIR', only=('INP', 'OUTP'), case=False

`output_trigger3_type`

str:

Constraints:
key='OUTP:TRIG3:OTYP', only=('DEV', 'TARM', 'UDEF'), case=False

`read_termination`

end of line string to expect in query replies

Constraints:
cache=True

Type
str

`reference_level`

float (dB):

Constraints:
key='DISP:TRAC1:Y:RLEV'

reference_level_trace2

float (dB):

Constraints:
key='DISP:TRAC2:Y:RLEV'

reference_level_trace3

float (dB):

Constraints:
key='DISP:TRAC3:Y:RLEV'

reference_level_trace4

float (dB):

Constraints:
key='DISP:TRAC4:Y:RLEV'

reference_level_trace5

float (dB):

Constraints:
key='DISP:TRAC5:Y:RLEV'

reference_level_trace6

float (dB):

Constraints:
key='DISP:TRAC6:Y:RLEV'

resolution_bandwidth

float (Hz):

Constraints:
key='BAND'

resource

device address or URI

Constraints:
cache=True, allow_none=True

Type
str

status_byte

instrument status decoded from '*STB?'

Constraints:
sets=False

Type
dict

sweep_points

int:

Constraints:
key='SWE:POIN'

`sweep_time`

float (Hz):

Constraints:
key='SWE:TIME'

`sweep_time_window2`

float (Hz):

Constraints:
key='SENS2:SWE:TIME'

`write_termination`

end of line string to send after writes

Constraints:
cache=True

Type
str

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

Bases: `RohdeSchwarzFSWBase`

`amplitude_offset`

float (dB):

Constraints:
key='DISP:TRAC1:Y:RLEV:OFFS'

`amplitude_offset_trace2`

float (dB):

Constraints:
key='DISP:TRAC2:Y:RLEV:OFFS'

`amplitude_offset_trace3`

float (dB):

Constraints:
key='DISP:TRAC3:Y:RLEV:OFFS'

`amplitude_offset_trace4`

float (dB):

Constraints:
key='DISP:TRAC4:Y:RLEV:OFFS'

`amplitude_offset_trace5`

float (dB):

Constraints:
key='DISP:TRAC5:Y:RLEV:OFFS'

`amplitude_offset_trace6`

float (dB):

Constraints:
key='DISP:TRAC6:Y:RLEV:OFFS'

`channel_type`

str:

Constraints:
key='INST', only=(None, 'SAN', 'IQ', 'RTIM'), case=False

`concurrency`

True if the device supports threading

Constraints:
sets=False

Type
bool

`default_trace`

data trace number to use if unspecified

Constraints:
cache=True

Type
str

`default_window`

data window number to use if unspecified

Constraints:
cache=True

Type
str

`display_update`

bool:

Constraints:
key='SYST:DISP:UPD', remap={False: 'OFF', True: 'ON'}

`expected_channel_type`

which channel type to use

Constraints:
sets=False, cache=True, only=(None, 'SAN', 'IQ', 'RTIM'), allow_none=True

Type
str

`format`

str:

Constraints:
key='FORM', only=('ASC,0', 'REAL,32', 'REAL,64', 'REAL,16'), case=False

`frequency_center`

float (Hz):

Constraints:
key='FREQ:CENT'

`frequency_span`

float (Hz):

Constraints:
key='FREQ:SPAN'

`frequency_start`

float (Hz):

Constraints:
key='FREQ:START'

`frequency_stop`

float (Hz):

Constraints:
key='FREQ:STOP'

`identity`

identity string reported by the instrument

Constraints:
key='*IDN', sets=False, cache=True

Type
str

`initiate_continuous`

bool:

Constraints:
key='INIT:CONT', remap={False: '0', True: '1'}

`input_attenuation`

float:

Constraints:
key='INP:ATT'

`input_attenuation_auto`

bool:

Constraints:
key='INP:ATT:AUTO', remap={False: '0', True: '1'}

`input_preamplifier_enabled`

bool:

Constraints:
key='INP:GAIN:STATE', remap={False: '0', True: '1'}

`isopen`

is the backend ready?

Type
bool

`options`

options reported by the instrument

Constraints:
key='*OPT', sets=False, cache=True

Type
str

`output_trigger2_direction`

str:

Constraints:
key='OUTP:TRIG2:DIR', only=('INP', 'OUTP'), case=False

`output_trigger2_type`

str:

Constraints:
key='OUTP:TRIG2:OTYP', only=('DEV', 'TARM', 'UDEF'), case=False

`output_trigger3_direction`

str:

Constraints:
key='OUTP:TRIG3:DIR', only=('INP', 'OUTP'), case=False

`output_trigger3_type`

str:

Constraints:
key='OUTP:TRIG3:OTYP', only=('DEV', 'TARM', 'UDEF'), case=False

`read_termination`

end of line string to expect in query replies

Constraints:
cache=True

Type
str

`reference_level`

float (dB):

Constraints:
key='DISP:TRAC1:Y:RLEV'

`reference_level_trace2`

float (dB):

Constraints:
key='DISP:TRAC2:Y:RLEV'

`reference_level_trace3`

float (dB):

Constraints:
key='DISP:TRAC3:Y:RLEV'

`reference_level_trace4`

float (dB):

Constraints:
key='DISP:TRAC4:Y:RLEV'

`reference_level_trace5`

float (dB):

Constraints:
key='DISP:TRAC5:Y:RLEV'

`reference_level_trace6`

float (dB):

Constraints:
key='DISP:TRAC6:Y:RLEV'

`resolution_bandwidth`

float (Hz):

Constraints:
key='BAND'

`resource`

device address or URI

Constraints:
cache=True, allow_none=True

Type
str

`status_byte`

instrument status decoded from '*STB?'

Constraints:
sets=False

Type
dict

`sweep_points`

int:

Constraints:
key='SWE:POIN'

`sweep_time`

float (Hz):

Constraints:
key='SWE:TIME'

`sweep_time_window2`
: float (Hz):

  Constraints:
  key='SENS2:SWE:TIME'

`write_termination`
: end of line string to send after writes

  Constraints:
  cache=True

  Type
  str

`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`
: float (dB):

  Constraints:
  key='DISP:TRAC1:Y:RLEV:OFFS'

`amplitude_offset_trace2`
: float (dB):

  Constraints:
  key='DISP:TRAC2:Y:RLEV:OFFS'

`amplitude_offset_trace3`
: float (dB):

  Constraints:
  key='DISP:TRAC3:Y:RLEV:OFFS'

`amplitude_offset_trace4`
: float (dB):

  Constraints:
  key='DISP:TRAC4:Y:RLEV:OFFS'

`amplitude_offset_trace5`
: float (dB):

  Constraints:
  key='DISP:TRAC5:Y:RLEV:OFFS'

`amplitude_offset_trace6`
: float (dB):

  Constraints:
  key='DISP:TRAC6:Y:RLEV:OFFS'

`channel_type`

str:

Constraints:
key='INST', only=(None, 'SAN', 'IQ', 'RTIM'), case=False

`concurrency`

True if the device supports threading

Constraints:
sets=False

Type
bool

`default_trace`

data trace number to use if unspecified

Constraints:
cache=True

Type
str

`default_window`

data window number to use if unspecified

Constraints:
cache=True

Type
str

`display_update`

bool:

Constraints:
key='SYST:DISP:UPD', remap={False: 'OFF', True: 'ON'}

`expected_channel_type`

which channel type to use

Constraints:
sets=False, cache=True, only=(None, 'SAN', 'IQ', 'RTIM'), allow_none=True

Type
str

`format`

str:

Constraints:
key='FORM', only=('ASC,0', 'REAL,32', 'REAL,64', 'REAL,16'), case=False

`frequency_center`

float (Hz):

Constraints:
key='FREQ:CENT'

`frequency_span`

float (Hz):

Constraints:
key='FREQ:SPAN'

`frequency_start`

float (Hz):

Constraints:
key='FREQ:START'

`frequency_stop`

float (Hz):

Constraints:
key='FREQ:STOP'

`identity`

identity string reported by the instrument

Constraints:
key='*IDN', sets=False, cache=True

Type
str

`initiate_continuous`

bool:

Constraints:
key='INIT:CONT', remap={False: '0', True: '1'}

`input_attenuation`

float:

Constraints:
key='INP:ATT'

`input_attenuation_auto`

bool:

Constraints:
key='INP:ATT:AUTO', remap={False: '0', True: '1'}

`input_preamplifier_enabled`

bool:

Constraints:
key='INP:GAIN:STATE', remap={False: '0', True: '1'}

`isopen`

is the backend ready?

Type
bool

`options`

options reported by the instrument

Constraints:
key='*OPT', sets=False, cache=True

Type
str

`output_trigger2_direction`

str:

Constraints:
key='OUTP:TRIG2:DIR', only=('INP', 'OUTP'), case=False

`output_trigger2_type`

str:

Constraints:
key='OUTP:TRIG2:OTYP', only=('DEV', 'TARM', 'UDEF'), case=False

`output_trigger3_direction`

str:

Constraints:
key='OUTP:TRIG3:DIR', only=('INP', 'OUTP'), case=False

`output_trigger3_type`

str:

Constraints:
key='OUTP:TRIG3:OTYP', only=('DEV', 'TARM', 'UDEF'), case=False

`read_termination`

end of line string to expect in query replies

Constraints:
cache=True

Type
str

`reference_level`

float (dB):

Constraints:
key='DISP:TRAC1:Y:RLEV'

`reference_level_trace2`

float (dB):

Constraints:
key='DISP:TRAC2:Y:RLEV'

reference_level_trace3

float (dB):

Constraints:
key='DISP:TRAC3:Y:RLEV'

reference_level_trace4

float (dB):

Constraints:
key='DISP:TRAC4:Y:RLEV'

reference_level_trace5

float (dB):

Constraints:
key='DISP:TRAC5:Y:RLEV'

reference_level_trace6

float (dB):

Constraints:
key='DISP:TRAC6:Y:RLEV'

resolution_bandwidth

float (Hz):

Constraints:
key='BAND'

resource

device address or URI

Constraints:
cache=True, allow_none=True

Type
str

status_byte

instrument status decoded from '*STB?'

Constraints:
sets=False

Type
dict

sweep_points

int:

Constraints:
key='SWE:POIN'

sweep_time

float (Hz):

Constraints:
key='SWE:TIME'

`sweep_time_window2`

float (Hz):

Constraints:
key='SENS2:SWE:TIME'

`write_termination`

end of line string to send after writes

Constraints:
cache=True

Type
str

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`

float (dB):

Constraints:
key='DISP:TRAC1:Y:RLEV:OFFS'

`amplitude_offset_trace2`

float (dB):

Constraints:
key='DISP:TRAC2:Y:RLEV:OFFS'

`amplitude_offset_trace3`

float (dB):

Constraints:
key='DISP:TRAC3:Y:RLEV:OFFS'

`amplitude_offset_trace4`

float (dB):

Constraints:
key='DISP:TRAC4:Y:RLEV:OFFS'

`amplitude_offset_trace5`

float (dB):

Constraints:
key='DISP:TRAC5:Y:RLEV:OFFS'

`amplitude_offset_trace6`

float (dB):

Constraints:
key='DISP:TRAC6:Y:RLEV:OFFS'

`channel_type`
str:

Constraints:
key='INST', only=(None, 'SAN', 'IQ', 'RTIM'), case=False

`concurrency`
True if the device supports threading

Constraints:
sets=False

Type
bool

`default_trace`
data trace number to use if unspecified

Constraints:
cache=True

Type
str

`default_window`
data window number to use if unspecified

Constraints:
cache=True

Type
str

`display_update`
bool:

Constraints:
key='SYST:DISP:UPD', remap={False: 'OFF', True: 'ON'}

`expected_channel_type`
which channel type to use

Constraints:
sets=False, cache=True, only=(None, 'SAN', 'IQ', 'RTIM'), allow_none=True

Type
str

`format`
str:

Constraints:
key='FORM', only=('ASC,0', 'REAL,32', 'REAL,64', 'REAL,16'), case=False

`frequency_center`

float (Hz):

Constraints:
key='FREQ:CENT'

`frequency_span`

float (Hz):

Constraints:
key='FREQ:SPAN'

`frequency_start`

float (Hz):

Constraints:
key='FREQ:START'

`frequency_stop`

float (Hz):

Constraints:
key='FREQ:STOP'

`identity`

identity string reported by the instrument

Constraints:
key='*IDN', sets=False, cache=True

Type
str

`initiate_continuous`

bool:

Constraints:
key='INIT:CONT', remap={False: '0', True: '1'}

`input_attenuation`

float:

Constraints:
key='INP:ATT'

`input_attenuation_auto`

bool:

Constraints:
key='INP:ATT:AUTO', remap={False: '0', True: '1'}

`input_preamplifier_enabled`

bool:

Constraints:
key='INP:GAIN:STATE', remap={False: '0', True: '1'}

`isopen`

is the backend ready?

Type
bool

`options`
options reported by the instrument

Constraints:
key='*OPT', sets=False, cache=True

Type
str

`output_trigger2_direction`
str:

Constraints:
key='OUTP:TRIG2:DIR', only=('INP', 'OUTP'), case=False

`output_trigger2_type`
str:

Constraints:
key='OUTP:TRIG2:OTYP', only=('DEV', 'TARM', 'UDEF'), case=False

`output_trigger3_direction`
str:

Constraints:
key='OUTP:TRIG3:DIR', only=('INP', 'OUTP'), case=False

`output_trigger3_type`
str:

Constraints:
key='OUTP:TRIG3:OTYP', only=('DEV', 'TARM', 'UDEF'), case=False

`read_termination`
end of line string to expect in query replies

Constraints:
cache=True

Type
str

`reference_level`
float (dB):

Constraints:
key='DISP:TRAC1:Y:RLEV'

`reference_level_trace2`
float (dB):

Constraints:
key='DISP:TRAC2:Y:RLEV'

`reference_level_trace3`

float (dB):

Constraints:
key='DISP:TRAC3:Y:RLEV'

`reference_level_trace4`

float (dB):

Constraints:
key='DISP:TRAC4:Y:RLEV'

`reference_level_trace5`

float (dB):

Constraints:
key='DISP:TRAC5:Y:RLEV'

`reference_level_trace6`

float (dB):

Constraints:
key='DISP:TRAC6:Y:RLEV'

`resolution_bandwidth`

float (Hz):

Constraints:
key='BAND'

`resource`

device address or URI

Constraints:
cache=True, allow_none=True

Type
str

`status_byte`

instrument status decoded from '*STB?'

Constraints:
sets=False

Type
dict

`sweep_points`

int:

Constraints:
key='SWE:POIN'

`sweep_time`

float (Hz):

Constraints:
key='SWE:TIME'

`sweep_time_window2`

float (Hz):

Constraints:
key='SENS2:SWE:TIME'

`write_termination`

end of line string to send after writes

Constraints:
cache=True

Type
str

`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`

float (dB):

Constraints:
key='DISP:TRAC1:Y:RLEV:OFFS'

`amplitude_offset_trace2`

float (dB):

Constraints:
key='DISP:TRAC2:Y:RLEV:OFFS'

`amplitude_offset_trace3`

float (dB):

Constraints:
key='DISP:TRAC3:Y:RLEV:OFFS'

`amplitude_offset_trace4`

float (dB):

Constraints:
key='DISP:TRAC4:Y:RLEV:OFFS'

`amplitude_offset_trace5`

float (dB):

Constraints:
key='DISP:TRAC5:Y:RLEV:OFFS'

`amplitude_offset_trace6`

float (dB):

Constraints:
key='DISP:TRAC6:Y:RLEV:OFFS'

`channel_type`

str:

Constraints:
key='INST', only=(None, 'SAN', 'IQ', 'RTIM'), case=False

`concurrency`

True if the device supports threading

Constraints:
sets=False

Type
bool

`default_trace`

data trace number to use if unspecified

Constraints:
cache=True

Type
str

`default_window`

data window number to use if unspecified

Constraints:
cache=True

Type
str

`display_update`

bool:

Constraints:
key='SYST:DISP:UPD', remap={False: 'OFF', True: 'ON'}

`expected_channel_type`

which channel type to use

Constraints:
sets=False, cache=True, only=(None, 'SAN', 'IQ', 'RTIM'), allow_none=True

Type
str

`format`

str:

Constraints:
key='FORM', only=('ASC,0', 'REAL,32', 'REAL,64', 'REAL,16'), case=False

`frequency_center`

float (Hz):

Constraints:
key='FREQ:CENT'

`frequency_span`

float (Hz):

Constraints:
key='FREQ:SPAN'

`frequency_start`

float (Hz):

Constraints:
key='FREQ:START'

`frequency_stop`

float (Hz):

Constraints:
key='FREQ:STOP'

`identity`

identity string reported by the instrument

Constraints:
key='*IDN', sets=False, cache=True

Type
str

`initiate_continuous`

bool:

Constraints:
key='INIT:CONT', remap={False: '0', True: '1'}

`input_attenuation`

float:

Constraints:
key='INP:ATT'

`input_attenuation_auto`

bool:

Constraints:
key='INP:ATT:AUTO', remap={False: '0', True: '1'}

`input_preamplifier_enabled`

bool:

Constraints:
key='INP:GAIN:STATE', remap={False: '0', True: '1'}

`isopen`

is the backend ready?

Type
bool

`options`

options reported by the instrument

Constraints:
key='*OPT', sets=False, cache=True

Type
str

`output_trigger2_direction`

str:

Constraints:
key='OUTP:TRIG2:DIR', only=('INP', 'OUTP'), case=False

`output_trigger2_type`

str:

Constraints:
key='OUTP:TRIG2:OTYP', only=('DEV', 'TARM', 'UDEF'), case=False

`output_trigger3_direction`

str:

Constraints:
key='OUTP:TRIG3:DIR', only=('INP', 'OUTP'), case=False

`output_trigger3_type`

str:

Constraints:
key='OUTP:TRIG3:OTYP', only=('DEV', 'TARM', 'UDEF'), case=False

`read_termination`

end of line string to expect in query replies

Constraints:
cache=True

Type
str

`reference_level`

float (dB):

Constraints:
key='DISP:TRAC1:Y:RLEV'

`reference_level_trace2`

float (dB):

Constraints:
key='DISP:TRAC2:Y:RLEV'

`reference_level_trace3`

float (dB):

Constraints:
key='DISP:TRAC3:Y:RLEV'

`reference_level_trace4`

float (dB):

Constraints:
key='DISP:TRAC4:Y:RLEV'

`reference_level_trace5`

float (dB):

Constraints:
key='DISP:TRAC5:Y:RLEV'

`reference_level_trace6`

float (dB):

Constraints:
key='DISP:TRAC6:Y:RLEV'

`resolution_bandwidth`

float (Hz):

Constraints:
key='BAND'

`resource`

device address or URI

Constraints:
cache=True, allow_none=True

Type
str

`status_byte`

instrument status decoded from '*STB?'

Constraints:
sets=False

Type
dict

`sweep_points`

int:

Constraints:
key='SWE:POIN'

`sweep_time`

float (Hz):

Constraints:
key='SWE:TIME'

`sweep_time_window2`

float (Hz):

Constraints:
key='SENS2:SWE:TIME'

`write_termination`

end of line string to send after writes

Constraints:
cache=True

Type
str

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`

float (dB):

Constraints:
key='DISP:TRAC1:Y:RLEV:OFFS'

`amplitude_offset_trace2`

float (dB):

Constraints:
key='DISP:TRAC2:Y:RLEV:OFFS'

`amplitude_offset_trace3`

float (dB):

Constraints:
key='DISP:TRAC3:Y:RLEV:OFFS'

`amplitude_offset_trace4`

float (dB):

Constraints:
key='DISP:TRAC4:Y:RLEV:OFFS'

`amplitude_offset_trace5`

float (dB):

Constraints:
key='DISP:TRAC5:Y:RLEV:OFFS'

`amplitude_offset_trace6`

float (dB):

Constraints:
key='DISP:TRAC6:Y:RLEV:OFFS'

`channel_type`

str:

Constraints:
key='INST', only=(None, 'SAN', 'IQ', 'RTIM'), case=False

`concurrency`

True if the device supports threading

Constraints:
sets=False

Type
bool

`default_trace`

data trace number to use if unspecified

Constraints:
cache=True

Type
str

`default_window`

data window number to use if unspecified

Constraints:
cache=True

Type
str

`display_update`

bool:

Constraints:
key='SYST:DISP:UPD', remap={False: 'OFF', True: 'ON'}

`expected_channel_type`

which channel type to use

Constraints:
sets=False, cache=True, only=(None, 'SAN', 'IQ', 'RTIM'), allow_none=True

Type
str

`format`

str:

Constraints:
key='FORM', only=('ASC,0', 'REAL,32', 'REAL,64', 'REAL,16'), case=False

`frequency_center`

float (Hz):

Constraints:
key='FREQ:CENT'

`frequency_span`

float (Hz):

Constraints:
key='FREQ:SPAN'

`frequency_start`

float (Hz):

Constraints:
key='FREQ:START'

`frequency_stop`

float (Hz):

Constraints:
key='FREQ:STOP'

`identity`

identity string reported by the instrument

Constraints:
key='*IDN', sets=False, cache=True

Type
str

`initiate_continuous`

bool:

Constraints:
key='INIT:CONT', remap={False: '0', True: '1'}

`input_attenuation`

float:

Constraints:
key='INP:ATT'

`input_attenuation_auto`

bool:

Constraints:
key='INP:ATT:AUTO', remap={False: '0', True: '1'}

`input_preamplifier_enabled`

bool:

Constraints:
key='INP:GAIN:STATE', remap={False: '0', True: '1'}

`isopen`

is the backend ready?

Type
bool

`options`
options reported by the instrument

Constraints:
key='*OPT', sets=False, cache=True

Type
str

`output_trigger2_direction`
str:

Constraints:
key='OUTP:TRIG2:DIR', only=('INP', 'OUTP'), case=False

`output_trigger2_type`
str:

Constraints:
key='OUTP:TRIG2:OTYP', only=('DEV', 'TARM', 'UDEF'), case=False

`output_trigger3_direction`
str:

Constraints:
key='OUTP:TRIG3:DIR', only=('INP', 'OUTP'), case=False

`output_trigger3_type`
str:

Constraints:
key='OUTP:TRIG3:OTYP', only=('DEV', 'TARM', 'UDEF'), case=False

`read_termination`
end of line string to expect in query replies

Constraints:
cache=True

Type
str

`reference_level`
float (dB):

Constraints:
key='DISP:TRAC1:Y:RLEV'

`reference_level_trace2`
float (dB):

Constraints:
key='DISP:TRAC2:Y:RLEV'

`reference_level_trace3`

float (dB):

Constraints:
key='DISP:TRAC3:Y:RLEV'

`reference_level_trace4`

float (dB):

Constraints:
key='DISP:TRAC4:Y:RLEV'

`reference_level_trace5`

float (dB):

Constraints:
key='DISP:TRAC5:Y:RLEV'

`reference_level_trace6`

float (dB):

Constraints:
key='DISP:TRAC6:Y:RLEV'

`resolution_bandwidth`

float (Hz):

Constraints:
key='BAND'

`resource`

device address or URI

Constraints:
cache=True, allow_none=True

Type
str

`status_byte`

instrument status decoded from '*STB?'

Constraints:
sets=False

Type
dict

`sweep_points`

int:

Constraints:
key='SWE:POIN'

`sweep_time`

float (Hz):

Constraints:
key='SWE:TIME'

`sweep_time_window2`

float (Hz):

Constraints:
key='SENS2:SWE:TIME'

`write_termination`

end of line string to send after writes

Constraints:
cache=True

Type
str

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

Bases: *RohdeSchwarzNRPSeries*

`average_auto`

bool:

Constraints:
key='AVER:COUN:AUTO', remap={False: 'OFF', True: 'ON'}

`average_count`

int:

Constraints:
key='AVER:COUN'

`average_enable`

bool:

Constraints:
key='AVER', remap={False: 'OFF', True: 'ON'}

`concurrency`

True if the device supports threading

Constraints:
sets=False

Type
bool

`frequency`

calibration frequency

Constraints:
key='SENS:FREQ'

Type
float (Hz)

`function`

str:

Constraints:
key='SENS:FUNC', only=('POW:AVG', 'POW:BURS:AVG', 'POW:TSL:AVG', 'XTIM:POW', 'XTIM:POWer'), case=False

`identity`

identity string reported by the instrument

Constraints:
key='*IDN', sets=False, cache=True

Type
str

`initiate_continuous`

bool:

Constraints:
key='INIT:CONT', remap={False: 'OFF', True: 'ON'}

`isopen`

is the backend ready?

Type
bool

`options`

options reported by the instrument

Constraints:
key='*OPT', sets=False, cache=True

Type
str

`read_termination`

str:

`resource`

device address or URI

Constraints:
cache=True, allow_none=True

Type
str

`smoothing_enable`

bool:

Constraints:
key='SMO:STAT', gets=False, remap={False: 'OFF', True: 'ON'}

`status_byte`

instrument status decoded from '*STB?'

Constraints:
sets=False

Type
dict

`trace_average_count`
int:

Constraints:
key='TRAC:AVER:COUN'

`trace_average_enable`
bool:

Constraints:
key='TRAC:AVER', remap={False: 'OFF', True: 'ON'}

`trace_average_mode`
str:

Constraints:
key='TRAC:AVER:TCON', only=('MOV', 'REP'), case=False

`trace_offset_time`
float:

Constraints:
key='TRAC:OFFS:TIME'

`trace_points`
int:

Constraints:
key='SENSe:TRACe:POINTs', gets=False

`trace_realtime`
bool:

Constraints:
key='TRAC:REAL', remap={False: 'OFF', True: 'ON'}

`trace_time`
float:

Constraints:
key='TRAC:TIME'

`trigger_count`
help me

Constraints:
key='TRIG:COUN'

Type
int

`trigger_delay`
float:

Constraints:
key='TRIG:DELAY'

`trigger_holdoff`

float:

Constraints:
key='TRIG:HOLD'

`trigger_level`

float:

Constraints:
key='TRIG:LEV'

`trigger_source`

No trigger; IMM: Software; INT: Internal level trigger; EXT2: External trigger, 10 kOhm

Constraints:
key='TRIG:SOUR', only=('HOLD', 'IMM', 'INT', 'EXT', 'EXT1', 'EXT2', 'BUS', 'INT1'), case=False

Type
str

Type
'HOLD

`write_termination`

end of line string to send after writes

Constraints:
cache=True

Type
str

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

Bases: *RohdeSchwarzNRPSeries*

`average_auto`

bool:

Constraints:
key='AVER:COUN:AUTO', remap={False: 'OFF', True: 'ON'}

`average_count`

int:

Constraints:
key='AVER:COUN'

`average_enable`

bool:

Constraints:
key='AVER', remap={False: 'OFF', True: 'ON'}

`concurrency`

True if the device supports threading

Constraints:
sets=False

Type
bool

`frequency`

calibration frequency

Constraints:
key='SENS:FREQ'

Type
float (Hz)

`function`

str:

Constraints:
key='SENS:FUNC', only=('POW:AVG', 'POW:BURS:AVG', 'POW:TSL:AVG', 'XTIM:POW', 'XTIM:POWer'), case=False

`identity`

identity string reported by the instrument

Constraints:
key='*IDN', sets=False, cache=True

Type
str

`initiate_continuous`

bool:

Constraints:
key='INIT:CONT', remap={False: 'OFF', True: 'ON'}

`isopen`

is the backend ready?

Type
bool

`options`

options reported by the instrument

Constraints:
key='*OPT', sets=False, cache=True

Type
str

`read_termination`

str:

`resource`

device address or URI

Constraints:
cache=True, allow_none=True

Type
str

`smoothing_enable`

bool:

Constraints:
key='SMO:STAT', gets=False, remap={False: 'OFF', True: 'ON'}

`status_byte`

instrument status decoded from '*STB?'

Constraints:
sets=False

Type
dict

`trace_average_count`

int:

Constraints:
key='TRAC:AVER:COUN'

`trace_average_enable`

bool:

Constraints:
key='TRAC:AVER', remap={False: 'OFF', True: 'ON'}

`trace_average_mode`

str:

Constraints:
key='TRAC:AVER:TCON', only=('MOV', 'REP'), case=False

`trace_offset_time`

float:

Constraints:
key='TRAC:OFFS:TIME'

`trace_points`

int:

Constraints:
key='SENSe:TRACe:POINTs', gets=False

`trace_realtime`

bool:

Constraints:
key='TRAC:REAL', remap={False: 'OFF', True: 'ON'}

`trace_time`

float:

Constraints:
key='TRAC:TIME'

`trigger_count`

help me

Constraints:
key='TRIG:COUN'

Type
int

`trigger_delay`

float:

Constraints:
key='TRIG:DELAY'

`trigger_holdoff`

float:

Constraints:
key='TRIG:HOLD'

`trigger_level`

float:

Constraints:
key='TRIG:LEV'

`trigger_source`

No trigger; IMM: Software; INT: Internal level trigger; EXT2: External trigger, 10 kOhm

Constraints:
key='TRIG:SOUR', only=('HOLD', 'IMM', 'INT', 'EXT', 'EXT1', 'EXT2', 'BUS', 'INT1'), case=False

Type
str

Type
'HOLD

`write_termination`

end of line string to send after writes

Constraints:
cache=True

Type
str

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`

bool:

Constraints:
key='AVER:COUN:AUTO', remap={False: 'OFF', True: 'ON'}

`average_count`

int:

Constraints:
key='AVER:COUN'

`average_enable`

bool:

Constraints:
key='AVER', remap={False: 'OFF', True: 'ON'}

`concurrency`

True if the device supports threading

Constraints:
sets=False

Type
bool

`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`

float (Hz):

Constraints:
key='SENS:FREQ'

`function`

str:

Constraints:
key='SENS:FUNC', only=('POW:AVG', 'POW:BURS:AVG', 'POW:TSL:AVG', 'XTIM:POW', 'XTIM:POWer'), case=False

`identity`

identity string reported by the instrument

Constraints:
key='*IDN', sets=False, cache=True

Type
str

`initiate_continuous`

bool:

Constraints:
key='INIT:CONT', remap={False: 'OFF', True: 'ON'}

`isopen`

is the backend ready?

Type
bool

`options`

options reported by the instrument

Constraints:
key='*OPT', sets=False, cache=True

Type
str

`preset()`

sends '*RST' to reset the instrument to preset

`read_termination`

str:

`resource`

device address or URI

Constraints:
cache=True, allow_none=True

Type
str

`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
bool:

Constraints:
key='SMO:STAT', gets=False, remap={False: 'OFF', True: 'ON'}

status_byte
instrument status decoded from '*STB?'

Constraints:
sets=False

Type
dict

trace_average_count
int:

Constraints:
key='TRAC:AVER:COUN'

trace_average_enable
bool:

Constraints:
key='TRAC:AVER', remap={False: 'OFF', True: 'ON'}

trace_average_mode
str:

Constraints:
key='TRAC:AVER:TCON', only=('MOV', 'REP'), case=False

trace_offset_time
float:

Constraints:
key='TRAC:OFFS:TIME'

trace_points
int:

Constraints:
key='SENSe:TRACe:POINTs', gets=False

trace_realtime
bool:

Constraints:
key='TRAC:REAL', remap={False: 'OFF', True: 'ON'}

trace_time
float:

Constraints:
key='TRAC:TIME'

`trigger_count`

help me

Constraints:
key='TRIG:COUN'

Type
int

`trigger_delay`

float:

Constraints:
key='TRIG:DELAY'

`trigger_holdoff`

float:

Constraints:
key='TRIG:HOLD'

`trigger_level`

float:

Constraints:
key='TRIG:LEV'

`trigger_single()`

`trigger_source`

str:

Constraints:
key='TRIG:SOUR', only=('HOLD', 'IMM', 'INT', 'EXT', 'EXT1', 'EXT2', 'BUS', 'INT1'), case=False

`write_termination`

end of line string to send after writes

Constraints:
cache=True

Type
str

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

Bases: `VISADevice`

`concurrency`

True if the device supports threading

Constraints:
sets=False

Type
bool

`frequency_center`

float (Hz):

Constraints:
key=':freq'

`identity`

identity string reported by the instrument

Constraints:
key='*IDN', sets=False, cache=True

Type
str

`isopen`

is the backend ready?

Type
bool

`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`

options reported by the instrument

Constraints:
key='*OPT', sets=False, cache=True

Type
str

`read_termination`

end of line string to expect in query replies

Constraints:
cache=True

Type
str

`resource`

device address or URI

Constraints:
cache=True, allow_none=True

Type
str

`rf_output_enable`

bool:

Constraints:
key='OUTP', remap={False: '0', True: '1'}

`rf_output_power`

float (dBm):

Constraints:
key=':pow'

`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`

instrument status decoded from '*STB?'

Constraints:
sets=False

Type
dict

`write_termination`

end of line string to send after writes

Constraints:
cache=True

Type
str

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

Bases: `VISADevice`

A network analyzer.

Author: Audrey Puls

`clear`()

`concurrency`

True if the device supports threading

Constraints:
sets=False

Type
bool

`identity`

identity string reported by the instrument

Constraints:
key='*IDN', sets=False, cache=True

Type
str

`initiate_continuous`

bool:

Constraints:
key='INITiate1:CONTinuous:ALL', remap={True: 'ON', False: 'OFF'}

`isopen`

is the backend ready?

Type
bool

`options`

options reported by the instrument

Constraints:
key='*OPT', sets=False, cache=True

Type
str

`read_termination`

end of line string to expect in query replies

Constraints:
cache=True

Type
str

`resource`

device address or URI

Constraints:
cache=True, allow_none=True

Type
str

`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`

instrument status decoded from '*STB?'

Constraints:
sets=False

Type
dict

`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`

end of line string to send after writes

Constraints:
cache=True

Type
str

`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`

Data rate of the physical serial connection.

Type
int

`concurrency`

True if the device supports threading

Constraints:
sets=False

Type
bool

`dsrdtr`

True to enable hardware (DSR/DTR) flow control.

Type
bool

`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`

is the backend ready?

Type
bool

`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`

Parity in the physical serial connection.

Type
bytes

`query`(*command*)

`reset`()

End any currently running scenario, then rewind

`resource`

serial port string (COMnn in windows or /dev/xxxx in unix/Linux)

Type
str

`rewind`()

Rewind the current scenario to the beginning.

`rtscts`

True to enable hardware (RTS/CTS) flow control.

Type
bool

`run`()

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

`running`

bool:

Constraints:
sets=False

`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`

bytes:

Constraints:
sets=False, only=(b'no scenario', b'loading', b'ready', b'arming', b'armed', b'running', b'paused', b'ended'), case=False

`stopbits`

Number of stop bits, one of *[1, 1.5, or 2.]*.

Type
float

`timeout`

Max time to wait for a connection before raising TimeoutError.

Type
float

`utc_time`

bytes:

Constraints:
sets=False

`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`

Termination character to send after a write.

Type
bytes

`xonxoff`

True to enable software flow control.

Type
bool

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

send and receive simultaneously

Type
bool

binary_path

path to the file to run

Constraints:
cache=True, allow_none=True

Type
Path

bind

bind connection to specified IP

Constraints:
allow_none=True

Type
str

bit_rate

maximum bit rate, accepts KMG unit suffix; defaults 1Mbit/s UDP, no limit for TCP

Constraints:
allow_none=True

Type
str (bits/s)

buffer_size

buffer size when generating traffic

Type
int (bytes)

concurrency

True if the device supports threading

Constraints:
sets=False

Type
bool

`format`

data unit prefix in bits (k, m, g), bytes (K, M, G), or None for auto

Constraints:
only=('k', 'm', 'g', 'K', 'M', 'G'), allow_none=True

Type
str

`interval`

seconds between throughput reports

Type
float (s)

`isopen`

is the backend ready?

Type
bool

`mss`

minimum segment size=MTU-40, TCP only

Type
int (bytes)

`nodelay`

set True to use nodelay (TCP traffic only)

Type
bool

`number`

the number of bytes to transmit before quitting

Type
int

`port`

network port

Type
int

`profile`(*block=True*)

`read_stdout()`

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

`report_style`

"C" for DataFrame table output, None for formatted text

Constraints:
only=('C', None), allow_none=True

Type
str

`resource`

client host address (set None if server=True)

Constraints:
allow_none=True

Type
str

`server`

True to run as a server

Type
bool

`tcp_window_size`

window / socket size (default OS dependent?)

Type
int (bytes)

`time`

10)

Type
float

Type
send duration (s) before quitting (default

`timeout`

wait time after close before killing the process

Constraints:
cache=True

Type
float (s)

`udp`

if True, to use UDP instead of TCP

Type
bool

`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`

send and receive simultaneously

Type
bool

`binary_path`

path to the file to run

Constraints:
cache=True, allow_none=True

Type
Path

`bind`

bind connection to specified IP

Constraints:
allow_none=True

Type
str

`bit_rate`

maximum bit rate, accepts KMG unit suffix; defaults 1Mbit/s UDP, no limit for TCP

Constraints:
allow_none=True

Type
str (bits/s)

`buffer_size`

buffer size when generating traffic

Type
int (bytes)

`children = {}`

`client`

the ip address from which the client sends data

Type
str

`close()`

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

`concurrency`

True if the device supports threading

Constraints:
sets=False

Type
bool

`format`

data unit prefix in bits (k, m, g), bytes (K, M, G), or None for auto

Constraints:
only=('k', 'm', 'g', 'K', 'M', 'G'), allow_none=True

Type
str

`interval`

seconds between throughput reports

Type
float (s)

`isopen`

is the backend ready?

Type
bool

`kill()`

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

`mss`

minimum segment size=MTU-40, TCP only

Type
int (bytes)

`nodelay`

set True to use nodelay (TCP traffic only)

Type
bool

`number`

the number of bytes to transmit before quitting

Type
int

`open()`

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

`port`

network port

Type
int

`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`
: "C" for DataFrame table output, None for formatted text

  Constraints:
  only=('C', None), allow_none=True

  Type
  str

`resource`
: unused - use sender and receiver instead

  Constraints:
  sets=False

  Type
  str

`running`()
: Check whether a background process is running.

  Returns
  True if running, otherwise False

`server`
: the ip address where the server listens

  Type
  str

`tcp_window_size`
: window / socket size (default OS dependent?)

  Type
  int (bytes)

`time`

10)

  Type
  float

  Type
  send duration (s) before quitting (default

`timeout`
: wait time after close before killing the process

  Constraints:
  cache=True

  Type
  float (s)

udp

if True, to use UDP instead of TCP

Type
bool

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

send and receive simultaneously

Type
bool

binary_path

path to the file to run

Constraints:
cache=True, allow_none=True

Type
Path

bind

bind connection to specified IP

Constraints:
allow_none=True

Type
str

bit_rate

maximum bit rate, accepts KMG unit suffix; defaults 1Mbit/s UDP, no limit for TCP

Constraints:
allow_none=True

Type
str (bits/s)

buffer_size

buffer size when generating traffic

Type
int (bytes)

`concurrency`
: True if the device supports threading

  Constraints:
  sets=False

  Type
  bool

`format`
: data unit prefix in bits (k, m, g), bytes (K, M, G), or None for auto

  Constraints:
  only=('k', 'm', 'g', 'K', 'M', 'G'), allow_none=True

  Type
  str

`interval`
: seconds between throughput reports

  Type
  float (s)

`isopen`
: is the backend ready?

  Type
  bool

`kill`(*wait_time=3*)
: Kill the local process and the iperf process on the UE.

`mss`
: minimum segment size=MTU-40, TCP only

  Type
  int (bytes)

`nodelay`
: set True to use nodelay (TCP traffic only)

  Type
  bool

`number`
: the number of bytes to transmit before quitting

  Type
  int

`open()`
: Open an adb connection to the handset, copy the iperf binary onto the phone, and verify that iperf executes.

`port`
: network port

  Type
  int

`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`

str:

Constraints:
cache=True

`report_style`

"C" for DataFrame table output, None for formatted text

Constraints:
only=('C', None), allow_none=True

Type
str

`resource`

client host address (set None if server=True)

Constraints:
allow_none=True

Type
str

`server`

True to run as a server

Type
bool

`tcp_window_size`

window / socket size (default OS dependent?)

Type
int (bytes)

`time`

10)

Type
float

Type
send duration (s) before quitting (default

`timeout`

wait time after close before killing the process

Constraints:
cache=True

Type
float (s)

`udp`

if True, to use UDP instead of TCP

Type
bool

`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`

path to the file to run

Constraints:
cache=True, allow_none=True

Type
Path

`bind`
bind connection to specified IP

Constraints:
allow_none=True

Type
str

`bit_rate`
maximum bit rate, accepts KMG unit suffix; defaults 1Mbit/s UDP, no limit for TCP

Constraints:
allow_none=True

Type
str (bits/s)

`buffer_size`
buffer size when generating traffic

Type
int (bytes)

`concurrency`
True if the device supports threading

Constraints:
sets=False

Type
bool

`format`
data unit prefix in bits (k, m, g), bytes (K, M, G), or None for auto

Constraints:
only=('k', 'm', 'g', 'K', 'M', 'G'), allow_none=True

Type
str

`interval`
seconds between throughput reports

Type
float (s)

`isopen`
is the backend ready?

Type
bool

`json`
output data in JSON format

Type
bool

`mss`
minimum segment size=MTU-40, TCP only

Type
int (bytes)

`nodelay`
set True to use nodelay (TCP traffic only)

Type
bool

`number`
the number of bytes to transmit before quitting

Type
int

`port`
network port

Type
int

`resource`
client host address (set None if server=True)

Constraints:
allow_none=True

Type
str

`reverse`
run in reverse mode (server sends, client receives)

Type
bool

`server`
True to run as a server

Type
bool

`tcp_window_size`
window / socket size (default OS dependent?)

Type
int (bytes)

`time`

10)

Type
float

Type
send duration (s) before quitting (default

`timeout`

wait time after close before killing the process

Constraints:
cache=True

Type
float (s)

`udp`

if True, to use UDP instead of TCP

Type
bool

`zerocopy`

use a 'zero copy' method of sending data

Type
bool

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

Bases: `Win32ComDevice`

QXDM software wrapper

`cache_path`

directory for auto-saved isf files

Type
str

`close()`

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

`com_object`

the win32com object string

Constraints:
sets=False

Type
str

`concurrency`

True if the device supports threading

Constraints:
sets=False

Type
bool

`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`

connection timeout (s)

Type
float

`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`

is the backend ready?

Type
bool

`open`()

Connect to the win32 com object

`reconnect`()

`resource`

serial port number for the handset connection

Type
int

`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`

Build ID of software on the phone

Constraints:
key=’ue_build_id’

Type
str

`ue_esn`

Phone ESN

Constraints:
key='ue_esn'

Type
str

`ue_imei`

Phone IMEI

Constraints:
key='ue_imei'

Type
str

`ue_mode`

current state of the phone

Constraints:
key='ue_mode'

Type
str

`ue_model_number`

model number code

Constraints:
key='ue_model_number'

Type
str

`version`

str:

Constraints:
sets=False, cache=True

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`

the name of the network interface that will receive data

Type
str

`concurrency`

True if the device supports threading

Constraints:
sets=False

Type
bool

`delay`

wait time before profiling

Constraints:
cache=True

Type
float

`isopen`

is the backend ready?

Type
bool

`mss()`

`mtu()`

`port`

TCP or UDP port for networking, or 0 to let the operating system choose

Type
int

`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`

which of the server or the client does the receiving

Constraints:
only=('server', 'client')

Type
str

`resource`

skipd - use sender and receiver instead

Constraints:
cache=True

Type
str

`server`

the name of the network interface that will send data

Type
str

`sync_each`

synchronize the start times of the send and receive threads for each buffer at the cost of throughput

Type
bool

`tcp_nodelay`

set True to disable Nagle's algorithm

Type
bool

`timeout`

timeout before aborting the test

Constraints:
cache=True

Type
float

`wait_for_interfaces`(*timeout*)

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

Bases: `Device`

`channel`

int:

`concurrency`

True if the device supports threading

Constraints:
sets=False

Type
bool

`description`

str:

Constraints:
sets=False, cache=True

`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`

is the backend ready?

Type
bool

`isup`

bool:

Constraints:
sets=False

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

`open()`

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

`refresh()`

`resource`

nn:nn:nn:nn)

Constraints:
cache=True

Type
str

Type
interface name (from the OS) or MAC address (nn

`signal`

int:

Constraints:
sets=False

`ssid`

SSID of the AP for connection

Type
str

`state`

str:

Constraints:
sets=False

`timeout`

attempt AP connection for this long before raising ConnectionError

Constraints:
cache=True

Type
float (s)

`transmit_rate_mbps`

int:

Constraints:
sets=False

`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`

path to the file to run

Constraints:
cache=True, allow_none=True

Type
Path

`concurrency`

True if the device supports threading

Constraints:
sets=False

Type
bool

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

`get_wlan_ssids`(*interface*)

`interface`
: name of the interface to query

  Type
  str

`isopen`
: is the backend ready?

  Type
  bool

`only_bssid`
: gather only BSSID information

  Type
  bool

`resource`
: device address or URI

  Constraints:
  cache=True, allow_none=True

  Type
  str

`timeout`
: wait time after close before killing the process

  Constraints:
  cache=True

  Type
  float (s)

`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*.