RsInstrument Python

Release 1.122.0

Rohde & Schwarz

Dec 08, 2025

CONTENTS:

CHAPTER

ONE

REVISION HISTORY

1.1 RsInstrument

RsInstrument module provides convenient way of communicating with R&S instruments.

Basic Hello-World code:

```
from RsInstrument import *

instr = RsInstrument('TCPIP::192.168.56.101::hislip0', id_query=True, reset=True)
idn = instr.query_str('*IDN?')
print('Hello, I am: ' + idn)
```

Check out the full documentation on ReadTheDocs.

Our public Rohde&Schwarz Github repository hosts many examples using this library. If you're looking for examples with specific instruments, check out the ones for Oscilloscopes, Powersensors, Powersupplies, Spectrum Analyzers, Vector Network Analyzers.

1.1.1 Version history:

Version 1.122.0.120 (08.12.2025)

- Added __all__ variable to the package's __init__ file.
- Added package trove classifier "Topic :: Scientific/Engineering :: Instrument Drivers".

Version 1.121.0.119 (14.11.2025)

- Changed typing hints alternatives.Example 'int or bool' -> 'int | bool'.
- Corrected help texts.

Version 1.120.1.118 (27.10.2025)

- Fixed py.typed file name

Version 1.120.0.117 (20.10.2025)

- Added MCP server support.

- Removed support for Python 3.8 and 3.9

Version 1.110.0.116 (15.10.2025)

- Added py.typed file to the top package.
- Added optional parameter mixed_mode to the method go_to_local().
- Improved help for is_connection_active() method.
- Core 1.106.0.
- New package build process with pyproject.toml.

Version 1.102.1.113 (16.07.2025)

- Fixed VisaSession's clear_before_read().
- Core 1.105.1

Version 1.102.0.112 (02.06.2025)

- Corrected duplicate time-statistics methods from logger.
- Improved help.
- Core 1.105.0

Version 1.101.0.111 (27.05.2025)

- Added settings profile 'RadEsT' for R&S Automotive Radar Tester.
- Added settings 'EachCmdSuffix' settings option - use it for instruments that require CRLF at the end of each command.
- Added settings 'StripStringTrailingWhitespaces' - use it to strip white spaces from string query responses.
- Added settings 'LoggingRelativeTimeOfFirstEntry' (boolean) - set it to true to start the logging with 00:00:00.000 relative time.
- Added check_status() method for checking instrument errors and throwing exception.
- Added in ScpiLogger: set_time_offset_zero_on_first_entry() - call it to have the next log entry starting with 00:00:00.000 relative time.
- Core 1.104.0

Version 1.100.0.110 (09.04.2025)

- Changed the minimum Python requirement to 3.8 to assure pyvisa version > 1.13.
- Fixed Logger end time in relative time mode.
- Fixed bug with InstrErrorSuppressor for checking errors flag after the context has finished.
- Extended InstrErrorSuppressor to catch queries Timeout as StatusException.
- Fixes for backend pyvisa-py.
- Added method get_option_counts() for getting number of a certain K-option occurrences in the original option string.
- Fixes for Pycharm 2024.3 checks.
- Core 1.103.0

Version 1.90.0.108 (07.10.2024)

- Changed the minimum Python requirement to 3.7 to avoid SCPI Logger Regex error.

- Fixed VISA Timeout Error generations for NRP sessions.
- Added Instrument Options methods: has_instr_option(), has_instr_option_regex(), has_instr_option_k0(), add_instr_option(), remove_instr_option()

Version 1.82.1.106 (13.06.2024)

- Fixed failing 'import visa' statement for python > 3.10

Version 1.82.0.105 (07.06.2024)

- **Changes in ScpiLogger:**
  - info(), error() - changed the last parameter 'cmd' to optional
  - info_bin(), info_list() - changed the order of the last two parameters!!! 'cmd' moved to the end and made optional, to be compatible with 1.61.0
  - ContextManager VisaTimeoutSuppressor made more robust in case of exceptions inside the other exceptions.

Version 1.80.0.103 (27.05.2024)

- Core 1.90.0 with more robust _flush_junk_data() that tolerates read timeouts.
- Added Logger.log_info_replacer for customizing the logging info strings.
- Logger info strings 'Write string' and 'Query string' shortened to 'Write' and 'Query'

Version 1.70.0.102 (26.04.2024)

- To all query_str_list_xxx() methods, added non-mandatory parameter 'remove_blank_response'.
- Logger: added new variable to the format string: %SCPI_COMMAND%, where you can only log SCPI commands to your log data.
- Added Context-managers for ignoring errors and ignoring VISA Timeouts:

```
# Any Instrument error in the context is ignored
with io.instr_err_suppressor() as supp:
    io.write("*RSaT")
if supp.get_errors_occurred():
    print("Error(s) suppressed")

# Any Timeout error in the context is ignored
with io.visa_tout_suppressor(500) as supp:
    io.write("*IDaN")
if supp.get_timeout_occurred():
    print("VISA Timeout suppressed")
```

- Added to Utilities interface: query_str_list(), query_str_list_with_opc(), query_bool_list(), query_bool_list_with_opc().
- Added Utility functions: value_to_si_string(), size_to_kb_mb_gb_string().
- **Changed behaviour of the Conversion functions to list:**
  - str_to_float_list()
  - str_to_float_or_bool_list()
  - str_to_int_list()
  - str_to_int_or_bool_list()

- str_to_bool_list()

These functions previously returned a list of one element if the input value was whitespace-only string. Now, in such case they return empty list.

Version 1.61.0.101 (27.02.2024)

- Added settings profile 'XK41' for R&S Software Defined Radios.
- Added settings 'FirstCmds' where you can send the defined commands right after the init. Send more commands in a row with ';;' separator.
- Added settings 'EachCmdPrefix' - this prefix is added to each command sent to the instrument. Supported values are also 'lf', 'cr', 'tab'.

Version 1.60.0.100 (31.01.2024)

- Fixed SocketIo session for cases when the instrument connection is lost in the middle of reading a response.
- Fixed VisaPluginSocketIo read() method for cases where the session is lost. The method now generates exception in that case.
- Added settings OpcSyncQueryMechanism with changed default value to 'only_check_mav_err_queue'.
- Added settings 'OpcSyncQueryMechanism' with values: Standard, AlsoCheckMav, ClsOnly-CheckMavErrQueue, OnlyCheckMavErrQueue.

Version 1.55.0.99 (29.09.2023)

- Added logger convenient methods start() and stop().
- Added lock_resource() and unlock_resource() methods for device-site locking.
- Added Context-manager interface to the RsInstrument class. Now you can use it as follows:

```
with RsInstrument("TCPIP::192.168.1.101::hislip0") as io:
    io.reset()
```

Version 1.54.0.98 (27.06.2023)

- Added new options profile for ATS chambers.
- Added settings boolean token EachCmdAsQuery. Example: EachCmdAsQuery=True. Default: False.

Version 1.53.1.97 (28.03.2023)

- Fixed decoding custom Status Register bits.

Version 1.53.0.96 (18.10.2022)

- Improved mode where the instrument works with a session from another object.
- Silently ignoring invalid *IDN? string.
- Added new options profile 'Minimal' for non-SCPI-99 instruments.

Version 1.52.0.94 (28.09.2022)

- Fixed DisableOpcQuery=True settings effect.
- Increased DataChunkSize from 1E6 to 1E7 bytes.
- Improved robustness of the TerminationCharacter option value entry.
- Added new options profile for CMQ500: 'Profile=CMQ'.

Version 1.51.1.93 (09.09.2022)

- Fixed go_to_local() / go_to_remote() for VXI-capable sessions.

Version 1.51.0.92 (08.09.2022)

- Changed the accepted IDN? response to more permissive.
- Removed build number from the package version.
- Added constructor options boolean token VxiCapable. Example: VxiCapable=False. Default: True (coerced later to false based on a session type).
- Added methods go_to_remote() and go_to_local().
- Added methods file_exists() and get_file_size().

Version 1.50.0.90 (23.06.2022)

- Added relative timestamp to the logger.
- Added RsInstrument class variables for logging making it possible to define common target and reference timestamp for all instances.
- Logger stream entries are by default immediately flushed, making sure that the log is complete.
- Added time statistic methods get_total_execution_time(), get_total_time(), reset_time_statistics().

Version 1.24.0.83 (03.06.2022)

- Changed parsing of SYST:ERR? response to tolerate +0,"No Error" response.
- Added constructor options integer token OpenTimeout. Example: OpenTimeout=5000. Default: 0.
- Added constructor options boolean token ExclusiveLock. Example: ExclusiveLock=True. Default: False.

Version 1.23.0.82 (25.05.2022)

- Added stripping of trailing commas when parsing the idn response.
- If the Resource Manager does not find any default VISA implementation, it falls back to R&S VISA - relevant for LINUX and MacOS.
- Other typos and formatting corrections.
- Changed parsing of SYST:ERR? response to tolerate +0,"No Error" response.

Version 1.22.0.80 (21.04.2022)

- Added optional parameter timeout to reset().
- Added query list methods: query_str_list, query_str_list_with_opc, query_bool_list, query_bool_list_with_opc.
- Added query_str_stripped for stripping string responses of quotes.

Version 1.21.0.78 (15.03.2022)

- Added logging to UDP port (49200) to integrate with new R&S Instrument Control plugin for Pycharm.
- Improved documentation for logging and Simulation mode sessions.

Version 1.20.0.76 (19.11.2021)

- Fixed logging strings when device name was a substring of the resource name.

Version 1.19.0.75 (08.11.2021)

- Added setting profile for non-standard instruments. Example of the options string: options='Profile=hm8123'.

Version 1.18.0.73 (15.10.2021)

- Added correct conversion of strings with SI suffixes (e.g.: MHz, KHz, THz, GHz, ms, us) to float and integer.

Version 1.17.0.72 (31.08.2021)

- Changed default encoding of string<=>bin from utf-8 to charmap.
- Added settable encoding for the session. Property: RsInstrument.encoding.
- Fixed logging to console when switched on after init - the cached init entries are now properly flushed and displayed.

Version 1.16.0.69 (17.07.2021)

- Improved exception handling in cases where the instrument session is closed.

Version 1.15.0.68 (12.07.2021)

- Scpi logger time entries now support not only datetime tuples, but also float timestamps.
- Added query_all_errors_with_codes() - returning list of tuples (message: str, code: int).
- Added logger.log_status_check_ok property. This allows for skipping lines with 'Status check: OK'.

Version 1.14.0.65 (28.06.2021)

- Added SCPI Logger.
- Simplified constructor's options string format - removed DriverSetup=() syntax. Instead of "DriverSetup=(TerminationCharacter='n')", you use "TerminationCharacter='n'". The original format is still supported.
- Fixed calling SYST:ERR? even if STB? returned 0.
- Replaced @ni backend with @ivi for resource manager - this is necessary for the future pyvisa version 1.12+.

Version 1.13.0.63 (09.06.2021)

- Added methods reconnect(), is_connection_active().

Version 1.12.1.60 (01.06.2021)

- Fixed bug with error checking when events are defined.

Version 1.12.0.58 (03.05.2021)

- Changes in Core only.

Version 1.11.0.57 (18.04.2021)

- **Added aliases for the write_str… and query_str… methods:**
  - write() = write_str()
  - query() = query_str()
  - write_with_opc() = write_str_with_opc()
  - query_with_opc() = query_str_with_opc()

Version 1.9.1.54 (20.01.2021)

- query_opc() got additional non-mandatory parameter 'timeout'.
- Code changes only relevant for the auto-generated drivers.

Version 1.9.0.52 (29.11.2020)

- Added Thread-locking for sessions. Related new methods: get_lock(), assign_lock(), clear_lock().
- Added read-only property 'resource_name'.

Version 1.8.4.49 (13.11.2020)

- Changed Authors and copyright.
- Code changes only relevant for the auto-generated drivers.
- Extended Conversions method str_to_str_list() by parameter 'clear_one_empty_item' with default value False.

Version 1.8.3.46 (09.11.2020)

- Fixed parsing of the instrument errors when an error message contains two double quotes.

Version 1.8.2.45 (21.10.2020)

- Code changes only relevant for the auto-generated drivers.
- Added 'UND' to the list of float numbers that are represented as NaN.

Version 1.8.1.41 (11.10.2020)

- Fixed Python 3.8.5+ warnings.
- Extended documentation, added offline installer.
- Filled package's __init__ file with the exposed API. This simplifies the import statement.

Version 1.7.0.37 (01.10.2020)

- Replaced 'import visa' with 'import pyvisa' to remove Python 3.8 pyvisa warnings.
- Added option to set the termination characters for reading and writing. Until now, it was fixed to '\n' (Linefeed). Set it in the constructor 'options' string: DriverSetup=(TerminationCharacter = '\r'). Default value is still '\n'.
- Added static method RsInstrument.assert_minimum_version() raising assertion exception if the RsInstrument version does not fulfill at minimum the entered version.
- Added 'Hameg' to the list of supported instruments.

Version 1.6.0.32 (21.09.2020)

- Added documentation on readthedocs.org.
- Code changes only relevant for the auto-generated drivers.

Version 1.5.0.30 (17.09.2020)

- Added recognition of RsVisa library location for linux when using options string 'SelectVisa=rs'.
- Fixed bug in reading binary data 16 bit.

Version 1.4.0.29 (04.09.2020)

- Fixed error for instruments that do not support *OPT? query.

Version 1.3.0.28 (18.08.2020)

- Implemented SocketIO plugin which allows the remote-control without any VISA installation.
- Implemented finding resources as a static method of the RsInstrument class.

Version 1.2.0.25 (03.08.2020)

- Fixed reading of long strings for NRP-Zxx sessions.

Version 1.1.0.24 (16.06.2020)

- Fixed simulation mode switching.
- Added Repeated capability.

Version 1.0.0.21

- First released version.

CHAPTER

TWO

WELCOME TO THE RSINSTRUMENT PYTHON STEP-BY-STEP GUIDE

CHAPTER

THREE

INTRODUCTION

RsInstrument is a Python remote-control communication module for Rohde & Schwarz SCPI-based Test and Measurement Instruments. After reading this guide you will be convinced of its edge over other remote-control packages.

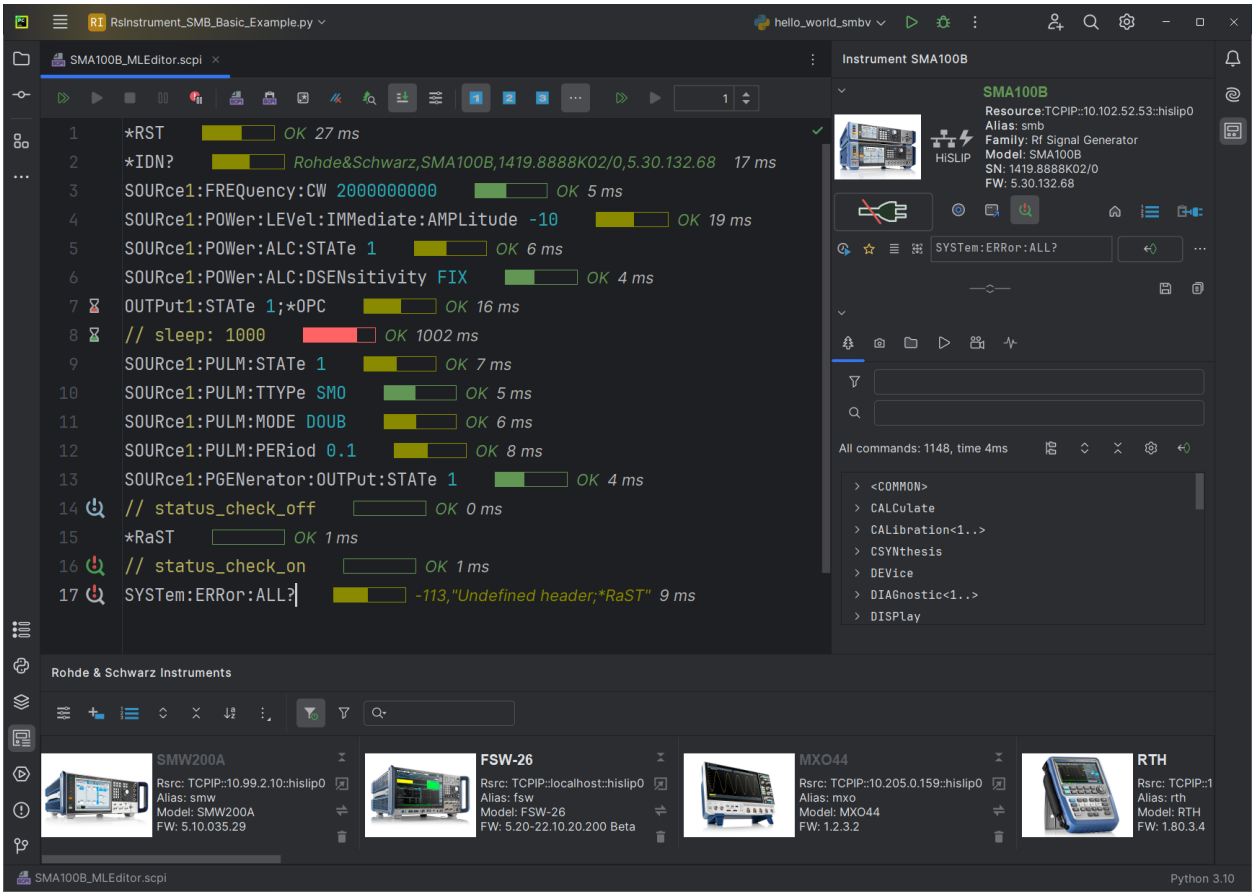
The original title of this document was **"10 Tips and Tricks…"**, but there were just too many neat features to fit into 10 chapters. Some of the RsInstrument's key features:

- Type-safe API using typing module
- You can select which VISA to use or even not use any VISA at all
- Initialization of a new session is straight-forward, no need to set any other properties
- Many useful features are already implemented - reset, self-test, opc-synchronization, error checking, option checking
- Binary data blocks transfer in both directions
- Transfer of arrays of numbers in binary or ASCII format
- File transfers in both directions
- Events generation in case of error, sent data, received data, chunk data
- Multithreading session locking - you can use multiple threads talking to one instrument at the same time
- Logging feature tailored for SCPI communication

Check out RsInstrument script examples here: Rohde & Schwarz GitHub Repository.

Short Getting-started video from our Oscilloscope guys:

Oh, one other thing - for Pycharm users we just released a Remote-control Plugin that makes your Pycharm development of remote-control script much faster:


RsInstrument_SMB_Basic_Example.py
hello_world_smbv
SMA100B_MLEditor.scpi
Instrument SMA100B
SMA100B
Resource:TCPIP::10.102.52.53::hislip0
Alias: smb
Family: Rf Signal Generator
Model: SMA100B
SN: 1419.8888K02/0
FW: 5.30.132.68
HiSLIP
SYSTem:ERRor:ALL?
All commands: 1148, time 4ms
<COMMON>
CALCulate
CALibration<1..>
CSYNthesis
DEVice
DIAGnostic<1..>
DISPlay
1 *RST OK 27 ms
2 *IDN? Rohde&Schwarz,SMA100B,1419.8888K02/0,5.30.132.68 17 ms
3 SOURce1:FREQuency:CW 2000000000 OK 5 ms
4 SOURce1:POWer:LEVel:IMMediate:AMPLitude -10 OK 19 ms
5 SOURce1:POWer:ALC:STATe 1 OK 6 ms
6 SOURce1:POWer:ALC:DSENsitivity FIX OK 4 ms
7 OUTPut1:STATe 1;*OPC OK 16 ms
8 // sleep: 1000 OK 1002 ms
9 SOURce1:PULM:STATe 1 OK 7 ms
10 SOURce1:PULM:TTYPe SMO OK 5 ms
11 SOURce1:PULM:MODE DOUB OK 6 ms
12 SOURce1:PULM:PERiod 0.1 OK 8 ms
13 SOURce1:PGENerator:OUTPut:STATe 1 OK 4 ms
14 // status_check_off OK 0 ms
15 *RaST OK 1 ms
16 // status_check_on OK 1 ms
17 SYSTem:ERRor:ALL? -113,"Undefined header;*RaST" 9 ms
Rohde & Schwarz Instruments
SMW200A
Rsrc: TCPIP::10.99.2.10::hislip0
Alias: smw
Model: SMW200A
FW: 5.10.035.29
FSW-26
Rsrc: TCPIP::localhost::hislip0
Alias: fsw
Model: FSW-26
FW: 5.20-22.10.20.200 Beta
MXO44
Rsrc: TCPIP::10.205.0.159::hislip0
Alias: mxo
Model: MXO44
FW: 1.2.3.2
RTH
Rsrc: TCPIP::1
Alias: rth
Model: RTH
FW: 1.80.3.4
SMA100B_MLEditor.scpi
Python 3.10

CHAPTER

FOUR

INSTALLATION

RsInstrument is hosted on pypi.org. You can install it with pip (for example `pip.exe` for Windows), or if you are using Pycharm (and you should be :-) direct in the Pycharm packet management GUI.

4.1 Option 1 - Installing with pip.exe under Windows

- Start the command console: `WinKey + R`, type `cmd` and hit ENTER
- Change the working directory to the Python installation of your choice (adjust the user name and python version in the path):

```
cd c:\Users\John\AppData\Local\Programs\Python\Python310\Scripts
```

- install RsInstrument with the command: `pip install RsInstrument`

4.2 Option 2 - Installing in Pycharm

- In Pycharm Menu `File->Settings->Project->Python Interpreter` click on the **'+'** button on the top left. Newer Pycharm versions have `Python Packages` Tool Window, you can perform the same operation there.
- Type `rsinstrument` in the search box
- Install the version 1.53.0 or newer
- If you are behind a Proxy server, configure it in the Menu: `File->Settings->Appearance->System Settings->HTTP Proxy`

For more information about Rohde & Schwarz instrument remote control, check out our Instrument remote control series: Rohde&Schwarz remote control Web series

4.3 Option 3 - Offline installation

If you are reading this, it is probably because none of the above worked for you - proxy problems, your boss saw the internet bill... Here are 5 easy steps for installing RsInstrument offline:

- Download this python script (**Save target as**): rsinstrument_offline_install.py
- Execute the script in your offline computer (supported is python 3.6 or newer)
- That's it ...
- Just watch the installation ...
- Enjoy ...

CHAPTER

FIVE

FINDING AVAILABLE INSTRUMENTS

Similar to the pyvisa's ResourceManager, RsInstrument can search for available instruments:

```
"""
Find the instruments in your environment.
"""

from RsInstrument import *

# Use the instr_list string items as resource names in the RsInstrument constructor
instr_list = RsInstrument.list_resources("?*")
print(instr_list)
```

If you have more VISAs installed, the one actually used by default is defined by a secret widget called VISA Conflict Manager. You can force your program to use a VISA of your choice:

```
"""
Find the instruments in your environment with the defined VISA implementation.
"""

from RsInstrument import *

# In the optional parameter visa_select you can use e.g.: 'rs' or 'ni'
# Rs Visa also finds any NRP-Zxx USB sensors
instr_list = RsInstrument.list_resources('?*', 'rs')
print(instr_list)
```

Tip

We believe our R&S VISA is the best choice for our customers. Couple of reasons why:

- Small footprint
- Superior VXI-11 and HiSLIP performance
- Integrated legacy sensors NRP-Zxx support
- Additional VXI-11 and LXI devices search
- Available for Windows, Linux, Mac OS

CHAPTER

SIX

INITIATING INSTRUMENT SESSION

RsInstrument offers four different types of starting your remote-control session. We begin with the most typical case, and progress with more special ones.

6.1 Standard Session Initialization

Initiating new instrument session happens, when you instantiate the RsInstrument object. Below, is a Hello World example. Different resource names are examples for different physical interfaces.

```
"""
Basic example on how to use the RsInstrument module for remote-controlling your VISA instrument.
Preconditions:
    - Installed RsInstrument Python module Version 1.50.0 or newer from pypi.org.
    - Installed VISA e.g. R&S Visa 5.12 or newer.
"""

from RsInstrument import *

# A good practice is to assure that you have a certain minimum version installed
RsInstrument.assert_minimum_version('1.102.0')
resource_string_1 = 'TCPIP::192.168.2.101::INSTR'  # Standard LAN connection (also called VXI-11)
resource_string_2 = 'TCPIP::192.168.2.101::hislip0'  # Hi-Speed LAN connection - see 1MA208
resource_string_3 = 'GPIB::20::INSTR'  # GPIB Connection
resource_string_4 = 'USB::0x0AAD::0x0119::022019943::INSTR'  # USB-TMC (Test and Measurement Class)
resource_string_5 = 'RSNRP::0x0095::104015::INSTR'  # R&S Powersensor NRP-Z86

# Initializing the session
instr = RsInstrument(resource_string_1)

idn = instr.query_str('*IDN?')
print(f"\nHello, I am: '{idn}'")
print(f'RsInstrument driver version: {instr.driver_version}')
print(f'Visa manufacturer: {instr.visa_manufacturer}')
print(f'Instrument full name: {instr.full_instrument_model_name}')
print(f'Instrument installed options: {",".join(instr.instrument_options)}')

```

(continues on next page)

(continued from previous page)

```
# Close the session
instr.close()
```

> **Note**
>
> If you are wondering about the `ASRL1::INSTR` - yes, it works too, but come on… it's 2024 :-)

Do not care about specialty of each session kind; RsInstrument handles all the necessary session settings for you. You have immediately access to many identification properties. Here are same of them:

```
idn_string: str
driver_version: str
visa_manufacturer: str
full_instrument_model_name: str
instrument_serial_number: str
instrument_firmware_version: str
instrument_options: List[str]
```

The constructor also contains optional boolean arguments `id_query` and `reset`:

```
instr = RsInstrument('TCPIP::192.168.56.101::hislip0', id_query=True, reset=True)
```

- Setting `id_query` to True (default is True) checks, whether your instrument can be used with the RsInstrument module.
- Setting `reset` to True (default is False) resets your instrument. It is equivalent to calling the `reset()` method.

If you tend to forget closing the session, use the context-manager. The session is closed even if the block inside `with` raises an exception:

```
"""
Using Context-Manager for you RsInstrument session.
No matter what happens inside the 'with' section, your session is always closed properly.
"""

from RsInstrument import *

RsInstrument.assert_minimum_version('1.102.0')
with RsInstrument('TCPIP::192.168.2.101::hislip0') as instr:
    idn = instr.query('*IDN?')
    print(f"\nHello, I am: '{idn}'")

```

6.2 Selecting specific VISA

Same as for the `list_resources()` function , RsInstrument allows you to choose which VISA to use:

```
"""
Choosing VISA implementation.
"""

```

(continues on next page)

(continued from previous page)

```
from RsInstrument import *

# Force use of the Rs Visa. For e.g.: NI Visa, use the "SelectVisa='ni'"
instr = RsInstrument('TCPIP::192.168.56.101::INSTR', True, True, "SelectVisa='rs'")

idn = instr.query_str('*IDN?')
print(f"\nHello, I am: '{idn}'")
print(f"\nI am using the VISA from: {instr.visa_manufacturer}")

# Close the session
instr.close()
```

6.3 No VISA Session

We recommend using VISA whenever possible, preferably with HiSLIP session because of its low latency. However, if you are a strict VISA-refuser, RsInstrument has something for you too:

No VISA raw LAN socket:

```
"""
Using RsInstrument without VISA for LAN Raw socket communication.
"""

from RsInstrument import *

instr = RsInstrument('TCPIP::192.168.56.101::5025::SOCKET', True, True, "SelectVisa=
↪'socket'")
print(f'Visa manufacturer: {instr.visa_manufacturer}')
print(f"\nHello, I am: '{instr.idn_string}'")
print(f"\nNo VISA has been harmed or even used in this example.")

# Close the session
instr.close()
```

> **Warning**
>
> Not using VISA can cause problems by debugging when you want to use the communication Trace Tool. The good news is, you can easily switch to use VISA and back just by changing the constructor arguments. The rest of your code stays unchanged.

6.4 Simulating Session

If a colleague is currently occupying your instrument, leave him in peace, and open a simulating session:

```
instr = RsInstrument('TCPIP::192.168.56.101::hislip0', True, True, "Simulate=True")
```

More `option_string` tokens are separated by comma:

```
instr = RsInstrument('TCPIP::192.168.56.101::hislip0', True, True, "SelectVisa='rs',␣
↪Simulate=True")
```

Note

Simulating session works as a database - when you write a command **SENSe:FREQ 10MHz**, the query **SENSe:FREQ?** returns **10MHz** back. For queries not preceded by set commands, the RsInstrument returns default values:

- **'Simulating'** for string queries.
- **0** for integer queries.
- **0.0** for float queries.
- **False** for boolean queries.

6.5 Shared Session

In some scenarios, you want to have two independent objects talking to the same instrument. Rather than opening a second VISA connection, share the same one between two or more RsInstrument objects:

```
"""
Sharing the same physical VISA session by two different RsInstrument objects.
"""

from RsInstrument import *

instr1 = RsInstrument('TCPIP::192.168.56.101::INSTR', True, True)
instr2 = RsInstrument.from_existing_session(instr1)

print(f'instr1: {instr1.idn_string}')
print(f'instr2: {instr2.idn_string}')

# Closing the instr2 session does not close the instr1 session - instr1 is the 'session master'
instr2.close()
print(f'instr2: I am closed now')

print(f'instr1: I am  still opened and working: {instr1.idn_string}')
instr1.close()
print(f'instr1: Only now I am closed.')
```

Note

The `instr1` is the object holding the 'master' session. If you call the `instr1.close()`, the `instr2` loses its instrument session as well, and becomes pretty much useless.

CHAPTER

SEVEN

CHECKING THE INSTALLED OPTIONS

If your instrument refuses to execute desired actions, and you do not know why - after all, the SCPI commands are in the User Manual, it's worth to check if the a special option is required. RsInstrument provides several ways to do it. The following example shows the literal CASE-INSENSITIVE searching:

```
"""
Checking the installed options with literal search.
"""

from RsInstrument import *

instr = RsInstrument('TCPIP::192.168.56.101::hislip0')

# Get all the options as list, and check for the specific one.
if 'K1' in instr.instrument_options:
        print('Option K1 installed')

# Check for one option.
# Keep in mind, that if the 'K0' is present, all the K-options are reported as installed.
if instr.has_instr_option('K1'):
        print('Option K1 installed')

# Check whether the K0 is installed:
if instr.has_instr_option_k0():
        print('You are a lucky customer, your instrument has all the K-options available.
↪')

# Check with a dedicated function for at least one option (logical OR).
if instr.has_instr_option('K1 / K1a / K1b'):
        print('At least one of the K1,K1a,K1b installed')

# Same as previous, but entered as a list of strings.
if instr.has_instr_option(['K1', 'K1a', 'K1b']):
        print('At least one of the K1,K1a,K1b installed')

instr.close()
```

Note

`instr.instrument_options` returns neatly sorted list of options, where the duplicates are removed, K-options are at the beginning, and the B-options at the end.

If for any reason you want to see how many times the K101 option was in the original Option's string, use the `io.get_option_counts('K101')`

Regular expressions CASE-INSENSITIVE searching. The Regex must be fully matched. That means, for example, `K1.` only positively matches `K10` or `K17`, but not `K1` or `K110`

```
"""
Checking the installed options with regular expressions search.
"""

from RsInstrument import *

instr = RsInstrument('TCPIP::192.168.56.101::hislip0')

# Check for one option.
# Keep in mind, that if the 'K0' is present, all the K-options are reported as installed.
if instr.has_instr_option_regex('K1.'):
    print('Option K10 or K11 or K12 up to K19 installed')

# Check with a dedicated function for at least one option (logical OR).
if instr.has_instr_option_regex('K1. / K2..'):
    print('At least one of the K10..K19, K200..K299 installed')

# Same as previous, but entered as a list of strings.
if instr.has_instr_option_regex(['K1.', 'K2..']):
    print('At least one of the K10..K19, K200..K299 installed')

instr.close()
```

If you wish to add or remove reported options, you can use `add_instr_option()` or `remove_instr_option()`. The `instr.instrument_options` is re-sorted after each change in the list:

```
"""
Changing how the installed options are reported.
This code does not actually install any option on the instrument :-)
"""

from RsInstrument import *

instr = RsInstrument('TCPIP::192.168.56.101::hislip0')

# I want to remove the 'K0' and see if the individual K-options are reported as present.
instr.remove_instr_option('K0')
if not instr.has_instr_option_k0():
    print('We have lost the K0, let us hope the individual options are still reported.')

instr.add_instr_option('K0')
if not instr.has_instr_option_k0():
    print('Now we have the K0 again :-)')
```

Note

It would be nice to install an instrument option with this small python script. Unfortunately, this is not the case - the script just manipulates the reported list of options. If you want to install an option on your instrument, you will have to buy it :-)

CHAPTER

EIGHT

BASIC I/O COMMUNICATION

Now we have opened the session, it's time to do some work. RsInstrument provides two basic methods for communication:

- `write()` - writing a command without an answer e.g.: ***RST**
- `query()` - querying your instrument, for example with the ***IDN?** query

Here, you may ask a question: Where is the `read()` ? Short answer - you do not need it. Long answer - your instrument never sends unsolicited responses. If you send a set-command, you use `write()`. For a query-command, you use `query()`. So, you really do not need it...

Enough with the theory, let us look at an example. Basic write, and query:

```
"""
Basic string write_str / query_str.
"""

from RsInstrument import *

instr = RsInstrument('TCPIP::192.168.56.101::INSTR', True, True)
instr.write_str('*RST')
response = instr.query_str('*IDN?')
print(response)

# Close the session
instr.close()
```

This example is so-called "*University-Professor-Example*" - good to show a principle, but never used in praxis. The previously mentioned commands are already a part of the driver's API. Here is another example, achieving the same goal:

```
"""
Basic string write_str / query_str.
"""

from RsInstrument import *

instr = RsInstrument('TCPIP::192.168.56.101::INSTR', True, True)
instr.reset()
print(instr.idn_string)

# Close the session
instr.close()
```

One additional feature we need to mention here: **VISA timeout**. To simplify, VISA timeout plays a role in each `query_xxx()`, where the controller (your PC) has to prevent waiting forever for an answer from your instrument. VISA timeout defines that maximum waiting time. You can set/read it with the `visa_timeout` property:

```
# Timeout in milliseconds
instr.visa_timeout = 3000
```

After this time, RsInstrument raises an exception. Speaking of exceptions, an important feature of the RsInstrument is **Instrument Status Checking**. Check out the next chapter that describes the error checking in details.

For completion, we mention other string-based `write_xxx()` and `query_xxx()` methods, all in one example. They are convenient extensions providing type-safe float/boolean/integer setting/querying features:

```
"""
Basic string write_xxx / query_xxx.
"""

from RsInstrument import *

instr = RsInstrument('TCPIP::192.168.56.101::INSTR', True, True)
instr.visa_timeout = 5000
instr.instrument_status_checking = True
instr.write_int('SWEEP:COUNT ', 10)  # sending 'SWEEP:COUNT 10'
instr.write_bool('SOURCE:RF:OUTPUT:STATE ', True)  # sending 'SOURCE:RF:OUTPUT:STATE ON'
instr.write_float('SOURCE:RF:FREQUENCY ', 1E9)  # sending 'SOURCE:RF:FREQUENCY 1000000000'

sc = instr.query_int('SWEEP:COUNT?')  # returning integer number sc=10
out = instr.query_bool('SOURCE:RF:OUTPUT:STATE?')  # returning boolean out=True
freq = instr.query_float('SOURCE:RF:FREQUENCY?')  # returning float number freq=1E9

# Close the session
instr.close()
```

Lastly, a method providing basic synchronization: `query_opc()`. It sends ***OPC?** to your instrument. The instrument waits with the answer until all the tasks it currently has in the execution queue are finished. This way your program waits too, and it is synchronized with actions in the instrument. Remember to have the VISA timeout set to an appropriate value to prevent the timeout exception. Here's a snippet:

```
instr.visa_timeout = 3000
instr.write("INIT")
instr.query_opc()

# The results are ready now to fetch
results = instr.query('FETCH:MEASUREMENT?')
```

You can define the VISA timeout directly in the `query_opc`, which is valid only for that call. Afterwards, the VISA timeout is set to the previous value:

```
instr.write("INIT")
instr.query_opc(3000)
```

Tip

Wait, there's more: you can send the ***OPC?** after each `write_xxx()` automatically:

```
# Default value after init is False
instr.opc_query_after_write = True
```

CHAPTER

NINE

ERROR CHECKING

RsInstrument has a built-in mechanism that after each command/query checks the instrument's status subsystem, and raises an exception if it detects an error. For those who are already screaming: **Speed Performance Penalty!!!**, don't worry, you can disable it.

Instrument status checking is very useful since in case your command/query caused an error, you are immediately informed about it. Status checking has in most cases no practical effect on the speed performance of your program. However, if for example, you do many repetitions of short write/query sequences, it might make a difference to switch it off:

```
# Default value after init is True
instr.instrument_status_checking = False
```

To clear the instrument status subsystem of all errors, call this method:

```
# Clear all the errors in the error queue
    instr.clear_status()
```

Instrument's status system error queue is clear-on-read. It means, if you query its content, you clear it at the same time. To query and clear list of all the current errors, use the following:

```
# Query all the errors in the error queue
    errors_list = instr.query_all_errors()
```

You can also check + clear the errors and raise exception if some errors occured:

```
# Check for errors and raise exception in case of one or more errors
instr.check_status()
```

See the next chapter on how to react on write/query errors.

9.1 Optimized Error Checking

As mentioned at the beginning of this chapter, there is a small performance penalty for checking errors after each command. This might play a bigger role if you are using many commands with short execution time, or repeat some measurement/setting in a loop. To benefit from error checking with minimal performance loss,try to follow this pattern:

- Keep the status checking ON for single, key commands.
- Switch the status checking OFF before a group of commands that logically belong together.
- Perform a group of write/query commands, for example a common configuration of a spectrum analyzer.
- After that, call `check_status()`. This method raises the `StatusException` (see Exceptions Handling Chapter below) if there are any errors in the error queue.

- Perform many SCPI write/query call in a loop.
- After the loop ends, perform `check_status()` again.

Let us see this in a practical example. Notice the emphasized lines 24, 31 and 45:

```
"""
How to optimize instrument status (error) checking.
Example contains commands for a spectrum analyzer.
"""

from RsInstrument import *

RsInstrument.assert_minimum_version('1.102.0')
instr = None
# Try-catch for initialization. If an error occurs, the ResourceError is raised
try:
        instr = RsInstrument('TCPIP::192.168.1.110::hislip0', True, True)
except ResourceError as e:
        print(e.args[0])
        print('Your instrument is probably OFF...')
        # Exit now, no point of continuing
        exit(1)

# Single commands, keep the instrument stats checking ON
instr.write('SYSTem:DISPlay:UPDate ON')
instr.write('INITiate:CONTinuous OFF')

# Switch the error checking off for a group of commands logically belonging together
instr.instrument_status_checking = False

# Configuration
instr.write('SENSe1:FREQuency:SPAN 10E6')
instr.write('SENSe1:BANDwidth:RESolution 1000')
instr.write('SENSe1:BANDwidth:VIDeo 100')
# Check status after this group of configuration commands
instr.check_status()

# Measure spectrum peaks in a loop
for freq in [10E6, 20E6, 100E6, 200E6, 500E6, 1E9]:
        instr.write_float('SENS:FREQ:CENT ', freq)
        instr.write_with_opc("INITiate:IMMediate")
        instr.write('CALC1:MARK1:STAT ON')
        instr.write('CALC1:MARK1:MAX:PEAK')
        marker_x = instr.query_float('CALCulate:MARKer1:X?')
        si_freq = value_to_si_string(freq)  # Nice way to create SI-formatted numbers
        si_marker_x = value_to_si_string(marker_x)
        marker_y = instr.query_float('CALCulate:MARKer1:Y?')
        print(f'Center Frequency {si_freq}Hz, peak: [{si_marker_x}Hz, {marker_y} dBm]')
        # Check status after one cycle of a marker measurement
        instr.check_status()

instr.close()
```

CHAPTER

TEN

EXCEPTION HANDLING

The base class for all the exceptions raised by the RsInstrument is `RsInstrException`. Inherited exception classes:

- `ResourceError` raised in the constructor by problems with initiating the instrument, for example wrong or non-existing resource name.
- `StatusException` raised if a command or a query generated error in the instrument's error queue.
- `TimeoutException` raised if a visa timeout or an opc timeout is reached.

In this example we show usage of all of them:

```
"""
How to deal with RsInstrument exceptions.
"""

from RsInstrument import *

RsInstrument.assert_minimum_version('1.102.0')
instr = None
# Try-catch for initialization. If an error occurs, the ResourceError is raised
try:
    instr = RsInstrument('TCPIP::192.168.1.110::hislip0', True, True)
except ResourceError as e:
    print(e.args[0])
    print('Your instrument is probably OFF...')
    # Exit now, no point of continuing
    exit(1)
try:
    # Dealing with commands that potentially generate instrument errors:
    # Switching the status checking OFF temporarily.
    # We use the InstrumentErrorSuppression context-manager that does it for us:
    with instr.instr_err_suppressor() as supp:
        instr.write('MY:MISSpelled:COMMand')
    if supp.get_errors_occurred():
        print("Errors occurred: ")
        for err in supp.get_all_errors():
            print(err)

    # Here for this query we use the reduced VISA timeout to prevent long waiting
    with instr.instr_err_suppressor(visa_tout_ms=500) as supp:
        idn = instr.query('*IDaN')
    if supp.get_errors_occurred():
```

(continues on next page)

(continued from previous page)

```
        print("Errors occurred: ")
        for err in supp.get_all_errors():
            print(err)

except StatusException as e:
    # Instrument status error
    print(e.args[0])
    print('Nothing to see here, moving on...')

except TimeoutException as e:
    # Timeout error
    print(e.args[0])
    print('That took a long time...')

except RsInstrException as e:
    # RsInstrException is a base class for all the RsInstrument exceptions
    print(e.args[0])
    print('Some other RsInstrument error...')

finally:
    instr.close()
```

10.1 Error Context Managers

You have seen in the example above the usage of two error Context-managers:

- Instrument status error Context-manager
- VISA timeout Context-manager

Instrument error suppressor Context-manager has several other neat features:

- It can change the VISA timeout for the commands in the context.
- It can selectively suppress only certain instrument error codes, and for others it raises exceptions.
- In case any other exception is raised within the context, the context-manager sets the VISA Timeout back to its original value.

Let us look at two examples. The following one only suppresses execution error (code -200). Since the command is misspelled, the error generated by the instrument has the code -113,'Undefined Header', and therefore the exception is raised anyway. This way you can only suppress certain errors. The context-manager object allows for checking if some errors were suppressed, and you can also read them all out:

```
"""
Suppress instrument errors with certain code with the Suppress Context-manager.
"""

from RsInstrument import *

RsInstrument.assert_minimum_version('1.102.0')
instr = RsInstrument('TCPIP::192.168.1.110::hislip0', True, True)

with instr.instr_err_suppressor(suppress_only_codes=-200) as supp:
```

(continues on next page)

(continued from previous page)

```
    # This will raise the exception anyway, because the Undefined Header error has code -113
    instr.write('MY:MISSpelled:COMMand')

if supp.get_errors_occurred():
    print("Errors occurred: ")
    for err in supp.get_all_errors():
        print(err)
```

You can also change the VISA Timeout inside the context:

```
with instr.instr_err_suppressor(visa_tout_ms=500, suppress_only_codes=-300) as supp:
            response = instr.query('*IDaN?')
```

Multiple error codes to suppress you enter as an integer list:

```
with instr.instr_err_suppressor(visa_tout_ms=3000, suppress_only_codes=[-200, -300]) as supp:
        meas = instr.query('MEASurement:RESult?')
```

If you are fighting with TimeoutExceptions, you'd like to react on them with a workaround, and continue with your code further, you have to do the following steps:

- adjust the VISA timeout to higher value to give the instrument more time, or to lower value to prevent long waiting times.
- execute the command / query.
- in case the timeout error occurrs, you clear the error queue to delete any 'Query Interrupted' errors.
- change the VISA timeout back to the original value.

This all is what the VISA Timeout Suppressor Context-manager does:

```
"""
Suppress VISA Timeout exception for certain commands with the Suppress Context-manager.
"""

from RsInstrument import *

RsInstrument.assert_minimum_version('1.102.0')
instr = RsInstrument('TCPIP::192.168.1.110::hislip0', True, True)

with instr.visa_tout_suppressor(visa_tout_ms=500) as supp:
    instr.query('*IDaN?')

if supp.get_timeout_occurred():
    print("Timeout occurred inside the context")
```

CHAPTER

ELEVEN

OPC-SYNCHRONIZED I/O COMMUNICATION

Now we are getting to the cool stuff: OPC-synchronized communication. OPC stands for OPeration Completed. The idea is: use one method (write or query), which sends the command, and polls the instrument's status subsystem until it indicates: **"I'm finished"**. The main advantage is, you can use this mechanism for commands that take several seconds, or minutes to complete, and you are still able to interrupt the process if needed. You can also perform other operations with the instrument in a parallel thread.

Now, you might say: **"This sounds complicated, I'll never use it"**. That is where the RsInstrument comes in: all the **write/query** methods we learned in the previous chapter have their `_with_opc` siblings. For example: `write()` has `write_with_opc()`. You can use them just like the normal write/query with one difference: They all have an optional parameter `timeout`, where you define the maximum time to wait. If you omit it, it uses a value from `opc_timeout` property. Important difference between the meaning of `visa_timeout` and `opc_timeout`:

- `visa_timeout` is a VISA IO communication timeout. **It does not play any role in the** `_with_opc()` methods. It only defines timeout for the standard `query_xxx()` methods. We recommend to keep it to maximum of 10000 ms.
- `opc_timeout` is a RsInstrument internal timeout, that serves as a default value to all the `_with_opc()` methods. If you explicitly define it in the method API, it is valid only for that one method call.

That was too much theory... now an example:

```
"""
Write / Query with OPC.
The SCPI commands syntax is for demonstration only.
"""

from RsInstrument import *

instr = RsInstrument('TCPIP::192.168.56.101::INSTR', True, True)
instr.visa_timeout = 3000
# opc_timeout default value is 10000 ms
instr.opc_timeout = 20000

# Send Reset command and wait for it to finish
instr.write_str_with_opc('*RST')

# Initiate the measurement and wait for it to finish, define the timeout 50 secs
# Notice no changing of the VISA timeout
instr.write_str_with_opc('INIT', 50000)
# The results are ready, simple fetch returns the results
# Waiting here is not necessary
result1 = instr.query_str('FETCH:MEASUREMENT?')
```

(continues on next page)

(continued from previous page)

```
22
23 # READ command starts the measurement, we use query_with_opc to wait for the measurement␣
   ↪to finish
24 result2 = instr.query_str_with_opc('READ:MEASUREMENT?', 50000)
25
26 # Close the session
27 instr.close()
```

CHAPTER

TWELVE

QUERYING ARRAYS

Often you need to query an array of numbers from your instrument, for example a spectrum analyzer trace or an oscilloscope waveform. Many programmers stick to transferring such arrays in ASCII format, because of the simplicity. Although simple, it is quite inefficient: one float 32-bit number can take up to 12 characters (bytes), compared to 4 bytes in a binary form. Well, with RsInstrument do not worry about the complexity: we have one method for binary or ascii array transfer.

12.1 Querying Float Arrays

Let us look at the example below. The method doing all the magic is `query_bin_or_ascii_float_list()`. In the 'waveform' variable, we get back a list of float numbers:

```
"""
Querying ASCII float arrays.
"""

from time import time
from RsInstrument import *

rto = RsInstrument('TCPIP::192.168.56.101::INSTR', True, True)
# Initiate a single acquisition and wait for it to finish
rto.write_str_with_opc("SINGle", 20000)

# Query array of floats in ASCII format
t = time()
waveform = rto.query_bin_or_ascii_float_list('FORM ASC;:CHAN1:DATA?')
print(f'Instrument returned {len(waveform)} points, query duration {time() - t:.3f} secs
↪')

# Close the RTO session
rto.close()
```

You might say: *I would do this with a simple 'query-string-and-split-on-commas'...* and you are right. The magic happens when we want the same waveform in binary form. One additional setting we need though - the binary data from the instrument does not contain information about its encoding. Is it 4 bytes float, or 8 bytes float? Low Endian or Big Endian? This, we specify with the property `bin_float_numbers_format`:

```
"""
Querying binary float arrays.
"""

```

(continues on next page)

(continued from previous page)

```
from RsInstrument import *
from time import time

rto = RsInstrument('TCPIP::192.168.56.101::INSTR', True, True)
# Initiate a single acquisition and wait for it to finish
rto.write_str_with_opc("SINGle", 20000)

# Query array of floats in Binary format
t = time()
# This tells the RsInstrument in which format to expect the binary float data
rto.bin_float_numbers_format = BinFloatFormat.Single_4bytes
# If your instrument sends the data with the swapped endianness, use the following format:
# rto.bin_float_numbers_format = BinFloatFormat.Single_4bytes_swapped
waveform = rto.query_bin_or_ascii_float_list('FORM REAL,32;:CHAN1:DATA?')
print(f'Instrument returned {len(waveform)} points, query duration {time() - t:.3f} secs')

# Close the RTO session
rto.close()
```

Tip

To find out in which format your instrument sends the binary data, check out the format settings: **FORM REAL,32** means floats, 4 bytes per number. It might be tricky to find out whether to swap the endianness. We recommend you simply try it out - there are only two options. If you see too many NaN values returned, you probably chose the wrong one:

- `BinFloatFormat.Single_4bytes` means the instrument and the control PC use the same endianness
- `BinFloatFormat.Single_4bytes_swapped` means they use opposite endiannesses

The same is valid for double arrays: settings **FORM REAL,64** corresponds to either `BinFloatFormat.Double_8bytes` or `BinFloatFormat.Double_8bytes_swapped`

12.2 Querying Integer Arrays

For performance reasons, we split querying float and integer arrays into two separate methods. The following example shows both ascii and binary array query. Here, the magic method is `query_bin_or_ascii_int_list()` returning list of integers:

```
"""
Querying ASCII and binary integer arrays.
"""

from RsInstrument import *
from time import time

rto = RsInstrument('TCPIP::192.168.56.101::INSTR', True, True)
# Initiate a single acquisition and wait for it to finish
rto.write_str_with_opc("SINGle", 20000)
```

(continues on next page)

(continued from previous page)

```
11
12 # Query array of integers in ASCII format
13 t = time()
14 waveform = rto.query_bin_or_ascii_int_list('FORM ASC;:CHAN1:DATA?')
15 print(f'Instrument returned {len(waveform)} points in ASCII format, query duration
   ↪{time() - t:.3f} secs')
16
17
18 # Query array of integers in Binary format
19 t = time()
20 # This tells the RsInstrument in which format to expect the binary integer data
21 rto.bin_int_numbers_format = BinIntFormat.Integer32_4bytes
22 # If your instrument sends the data with the swapped endianness, use the following␣
   ↪format:
23 # rto.bin_int_numbers_format = BinIntFormat.Integer32_4bytes_swapped
24 waveform = rto.query_bin_or_ascii_int_list('FORM INT,32;:CHAN1:DATA?')
25 print(f'Instrument returned {len(waveform)} points in binary format, query duration
   ↪{time() - t:.3f} secs')
26
27 # Close the rto session
28 rto.close()
```

CHAPTER

THIRTEEN

QUERYING BINARY DATA

A common question from customers: How do I read binary data to a byte stream, or a file?

If you want to transfer files between PC and your instrument, check out the following chapter: Transferring_Files.

13.1 Querying to bytes

Let us say you want to get raw (bytes) RTO waveform data. Call this method:

```
data = rto.query_bin_block('FORM REAL,32;:CHAN1:DATA?')
```

13.2 Querying to PC files

Modern instrument can acquire gigabytes of data, which is often more than your program can hold in memory. The solution may be to save this data to a file. RsInstrument is smart enough to read big data in chunks, which it immediately writes into a file stream. This way, at any given moment your program only holds one chunk of data in memory. You can set the chunk size with the property `data_chunk_size`. The initial value is 100 000 bytes.

We are going to read the RTO waveform into a PC file *c:\temp\rto_waveform_data.bin*:

```
rto.data_chunk_size = 10000
rto.query_bin_block_to_file(
    'FORM REAL,32;:CHAN1:DATA?',
    r'c:\temp\rto_waveform_data.bin',
    append=False)
```

CHAPTER

FOURTEEN

WRITING BINARY DATA

14.1 Writing from bytes data

We take an example for a Signal generator waveform data file. First, we construct a `wform_data` as `bytes`, and then send it with `write_bin_block()`:

```
# MyWaveform.wv is an instrument file name under which this data is stored
smw.write_bin_block("SOUR:BB:ARB:WAV:DATA 'MyWaveform.wv',", wform_data)
```

Note

Notice the `write_bin_block()` has two parameters:

- string parameter `cmd` for the SCPI command
- bytes parameter `payload` for the actual data to send

14.2 Writing from PC files

Similar to querying binary data to a file, you can write binary data from a file. The second parameter is the source PC file path with content which you want to send:

```
smw.write_bin_block_from_file("SOUR:BB:ARB:WAV:DATA 'MyWaveform.wv',", r"c:\temp\wform_
↪data.wv")
```

CHAPTER

FIFTEEN

TRANSFERRING FILES

15.1 Instrument -> PC

You just did a perfect measurement, saved the results as a screenshot to the instrument's storage drive. Now you want to transfer it to your PC. With RsInstrument, no problem, just figure out where the screenshot was stored on the instrument. In our case, it is *var/user/instr_screenshot.png*:

```
instr.read_file_from_instrument_to_pc(
    r'/var/user/instr_screenshot.png',
    r'c:\temp\pc_screenshot.png')
```

15.2 PC -> Instrument

Another common scenario: Your cool test program contains a setup file you want to transfer to your instrument: Here is the RsInstrument one-liner split into 3 lines:

```
instr.send_file_from_pc_to_instrument(
    'c:\MyCoolTestProgram\instr_setup.sav',
    r'/var/appdata/instr_setup.sav')
```

Tip

You want to delete a file on the instrument, but the instrument reports an **error**, because the file does not exist?
Or you want to write a file to the instrument, but get an **error** that the file already exists and can not be overwritten?
Not anymore, use the file detection methods:

```
# Do you exist?
i_exist = instr.file_exist(r'/var/appdata/instr_setup.sav')

# Give me your size or give me nothing...
your_size = instr.get_file_size(r'/var/appdata/instr_setup.sav')
```

CHAPTER

SIXTEEN

TRANSFERRING BIG DATA WITH PROGRESS

We can agree that it can be annoying using an application that shows no progress for long-lasting operations. The same is true for remote-control programs. Luckily, RsInstrument has this covered. And, this feature is quite universal - not just for big files transfer, but for any data in both directions.

RsInstrument allows you to register a function (programmer's fancy name is `handler` or `callback`), which is then periodically invoked after transfer of one data chunk. You can define that chunk size, which gives you control over the callback invoke frequency. You can even slow down the transfer speed, if you want to process the data as they arrive (direction instrument -> PC).

To show this in praxis, we are going to use another *University-Professor-Example*: querying the ***IDN?** with chunk size of 2 bytes and delay of 200ms between each chunk read:

```
"""
Event handlers by reading.
"""

from RsInstrument import *
import time


def my_transfer_handler(args):
    """Function called each time a chunk of data is transferred"""
    # Total size is not always known at the beginning of the transfer
    total_size = args.total_size if args.total_size is not None else "unknown"

    print(f"Context: '{args.context}{'with opc' if args.opc_sync else ''}', "
            f"chunk {args.chunk_ix}, "
            f"transferred {args.transferred_size} bytes, "
            f"total size {total_size}, "
            f"direction {'reading' if args.reading else 'writing'}, "
            f"data '{args.data}'")

    if args.end_of_transfer:
        print('End of Transfer')
    time.sleep(0.2)


instr = RsInstrument('TCPIP::192.168.56.101::INSTR', True, True)

instr.events.on_read_handler = my_transfer_handler
# Switch on the data to be included in the event arguments
```

(continues on next page)

(continued from previous page)

```
# The event arguments args.data will be updated
instr.events.io_events_include_data = True
# Set data chunk size to 2 bytes
instr.data_chunk_size = 2
instr.query_str('*IDN?')
# Unregister the event handler
instr.events.on_read_handler = None

# Close the session
instr.close()
```

If you start it, you might wonder (or maybe not): why is the `args.total_size = None`? The reason is, in this particular case the RsInstrument does not know the size of the complete response up-front. However, if you use the same mechanism for transfer of a known data size (for example, a file transfer), you get the information about the total size too, and hence you can calculate the progress as:

*progress [pct] = 100 * args.transferred_size / args.total_size*

Snippet of transferring file from PC to instrument, the rest of the code is the same as in the previous example:

```
instr.events.on_write_handler = my_transfer_handler
instr.events.io_events_include_data = True
instr.data_chunk_size = 1000
instr.send_file_from_pc_to_instrument(
    r'c:\MyCoolTestProgram\my_big_file.bin',
    r'/var/user/my_big_file.bin')
# Unregister the event handler
instr.events.on_write_handler = None
```

CHAPTER

SEVENTEEN

MULTITHREADING

You are at the party, many people talking over each other. Not every person can deal with such crosstalk, neither can measurement instruments. For this reason, RsInstrument has a feature of scheduling the access to your instrument by using so-called **Locks**. Locks make sure that there can be just one client at a time 'talking' to your instrument. Talking in this context means completing one communication step - one command write or write/read or write/read/error check.

To describe how it works, and where it matters, we take three typical multithread scenarios:

17.1 One instrument session, accessed from multiple threads

You are all set - the lock is a part of your instrument session. Check out the following example - it will execute properly, although the instrument gets 10 queries at the same time:

```
"""
Multiple threads are accessing one RsInstrument object.
"""

import threading
from RsInstrument import *


def execute(session: RsInstrument) -> None:
    """Executed in a separate thread."""
    session.query_str('*IDN?')


RsInstrument.assert_minimum_version('1.102.0')
instr = RsInstrument('TCPIP::192.168.56.101::INSTR')
threads = []
for i in range(10):
    t = threading.Thread(target=execute, args=(instr, ))
    t.start()
    threads.append(t)
print('All threads started')

# Wait for all threads to join this main thread
for t in threads:
    t.join()
print('All threads ended')

instr.close()
```

17.2 Shared instrument session, accessed from multiple threads

Same as in the previous case, you are all set. The session carries the lock with it. You have two objects, talking to the same instrument from multiple threads. Since the instrument session is shared, the same lock applies to both objects causing the exclusive access to the instrument.

Try the following example:

```
"""
Multiple threads are accessing two RsInstrument objects with shared session.
"""

import threading
from RsInstrument import *


def execute(session: RsInstrument, session_ix, index) -> None:
    """Executed in a separate thread."""
    print(f'{index} session {session_ix} query start...')
    session.query_str('*IDN?')
    print(f'{index} session {session_ix} query end')


RsInstrument.assert_minimum_version('1.102.0')
instr1 = RsInstrument('TCPIP::192.168.56.101::INSTR')
instr2 = RsInstrument.from_existing_session(instr1)
instr1.visa_timeout = 200
instr2.visa_timeout = 200
# To see the effect of crosstalk, uncomment this line
# instr2.clear_lock()

threads = []
for i in range(10):
    t = threading.Thread(target=execute, args=(instr1, 1, i,))
    t.start()
    threads.append(t)
    t = threading.Thread(target=execute, args=(instr2, 2, i,))
    t.start()
    threads.append(t)
print('All threads started')

# Wait for all threads to join this main thread
for t in threads:
    t.join()
print('All threads ended')

instr2.close()
instr1.close()
```

As you see, everything works fine. If you want to simulate some party crosstalk, uncomment the line `instr2.clear_lock()`. This causes the instr2 session lock to break away from the instr1 session lock. Although the instr1 still tries to schedule its instrument access, the instr2 tries to do the same at the same time, which leads to all the fun stuff happening.

17.3 Multiple instrument sessions accessed from multiple threads

Here, there are two possible scenarios depending on the instrument's capabilities:

- You are lucky, because you instrument handles each remote session completely separately. An example of such instrument is SMW200A. In this case, you have no need for session locking.
- Your instrument handles all sessions with one set of in/out buffers. You need to lock the session for the duration of a talk. And you are lucky again, because the RsInstrument takes care of it for you. The text below describes this scenario.

Run the following example:

```
"""
Multiple threads are accessing two RsInstrument objects with two separate sessions.
"""

import threading
from RsInstrument import *


def execute(session: RsInstrument, session_ix, index) -> None:
    """Executed in a separate thread."""
    print(f'{index} session {session_ix} query start...')
    session.query_str('*IDN?')
    print(f'{index} session {session_ix} query end')


RsInstrument.assert_minimum_version('1.102.0')
instr1 = RsInstrument('TCPIP::192.168.56.101::INSTR')
instr2 = RsInstrument('TCPIP::192.168.56.101::INSTR')
instr1.visa_timeout = 200
instr2.visa_timeout = 200

# Synchronise the sessions by sharing the same lock
instr2.assign_lock(instr1.get_lock())  # To see the effect of crosstalk, comment this␣
↪line

threads = []
for i in range(10):
    t = threading.Thread(target=execute, args=(instr1, 1, i,))
    t.start()
    threads.append(t)
    t = threading.Thread(target=execute, args=(instr2, 2, i,))
    t.start()
    threads.append(t)
print('All threads started')

# Wait for all threads to join this main thread
for t in threads:
    t.join()
print('All threads ended')

instr2.close()
instr1.close()
```

You have two completely independent sessions that want to talk to the same instrument at the same time. This will not go well, unless they share the same session lock. The key command to achieve this is `instr2.assign_lock(instr1.get_lock())` Comment that line, and see how it goes. If despite commenting the line the example runs without issues, you are lucky to have an instrument similar to the SMW200A.

CHAPTER

EIGHTEEN

LOGGING

Yes, the logging again. This one is tailored for instrument communication. You will appreciate such handy feature when you troubleshoot your program, or just want to protocol the SCPI communication for your test reports.

What can you do with the logger?

- Write SCPI communication to a stream-like object, for example console or file, or both simultaneously
- Log only errors and skip problem-free parts; this way you avoid going through thousands lines of texts
- Investigate duration of certain operations to optimize your program's performance
- Log custom messages from your program

The logged information can be sent to these targets (one or multiple):

- **Console**: this is the most straight-forward target, but it mixes up with other program outputs...
- **Stream**: the most universal one, see the examples below.
- **UDP Port**: if you wish to send it to another program, or a universal UDP listener. This option is used for example by our Instrument Control Pycharm Plugin.

18.1 Logging to console

```
"""
Basic logging example to the console.
"""

from RsInstrument import *

RsInstrument.assert_minimum_version('1.102.0')
instr = RsInstrument('TCPIP::192.168.1.101::INSTR')

# Switch ON logging to the console.
instr.logger.log_to_console = True
instr.logger.start()
instr.reset()

# Close the session
instr.close()
```

Console output:

```
10:29:10.819     TCPIP::192.168.1.101::INSTR     0.976 ms  Write: *RST
10:29:10.819     TCPIP::192.168.1.101::INSTR  1884.985 ms  Status check: OK
10:29:12.704     TCPIP::192.168.1.101::INSTR     0.983 ms  Query OPC: 1
10:29:12.705     TCPIP::192.168.1.101::INSTR     2.892 ms  Clear status: OK
10:29:12.708     TCPIP::192.168.1.101::INSTR     3.905 ms  Status check: OK
10:29:12.712     TCPIP::192.168.1.101::INSTR     1.952 ms  Close: Closing session
```

The columns of the log are aligned for better reading. Columns meaning:

(1) Start time of the operation.

(2) Device resource name. You can set an alias.

(3) Duration of the operation.

(4) Log entry.

Tip

You can customize the logging format with `set_format_string()`, and set the maximum log entry length with these properties:

- `abbreviated_max_len_ascii`
- `abbreviated_max_len_bin`
- `abbreviated_max_len_list`

See the full logger help *here*.

Notice the SCPI communication starts from the line `instr.reset()`. If you want to log the initialization of the session as well, you have to switch the logging ON already in the constructor:

```
instr = RsInstrument('TCPIP::192.168.56.101::hislip0', options='LoggingMode=On')
```

Note

`instr.logger.start()` and `instr.logger.mode = LoggingMode=On` have the same effect. However, in the constructor's options string, you can only use the `LoggingMode=On` format.

18.2 Logging to files

Parallel to the console logging, you can log to a general stream. Do not fear the programmer's jargon'... under the term **stream** you can just imagine a file. To be a little more technical, a stream in Python is any object that has two methods: `write()` and `flush()`. This example opens a file and sets it as logging target:

```
"""
Example of logging to a file.
"""

from RsInstrument import *

RsInstrument.assert_minimum_version('1.102.0')
instr = RsInstrument('TCPIP::192.168.1.101::INSTR')
```

(continues on next page)

(continued from previous page)

```

# We also want to log to the console.
instr.logger.log_to_console = True

# Logging target is our file
file = open(r'c:\temp\my_file.txt', 'w')
instr.logger.set_logging_target(file)
instr.logger.start()

# Instead of the 'TCPIP::192.168.1.101::INSTR', show 'MyDevice'
instr.logger.device_name = 'MyDevice'

# Custom user entry
instr.logger.info_raw('----- This is my custom log entry. ---- ')

instr.reset()

# Close the session
instr.close()

# Close the log file
file.close()
```

18.3 Logging with 00:00:00.000 start time

Very often, you do not need the absolute time in logging, but rather the relative times from the beginning. This way you quickly see the duration of you procedure. To set this up, use the constructor's option string `LoggingRelativeTimeOfFirstEntry=True` or the method `set_time_offset_zero_on_first_entry()`. Another nice feature is time statistic - instrument execution time and total time. These work independent from the fact, whether the logger is running or not:

```
"""
Logging example with:
 - Logging to the console.
 - Initialization in the constructor.
 - Time will be relative to the first log entry.
 - Time statistics at the end.
"""
import time

from RsInstrument import *

RsInstrument.assert_minimum_version('1.102.0')
instr = RsInstrument(
    'TCPIP::192.168.1.101::hislip0',
    options='LoggingToConsole=True, LoggingMode=On, LoggingRelativeTimeOfFirstEntry=True
↪')

print('\nEntries above come from the constructor.\n')
idn = instr.query('*IDN?')
# Pause for 1 second, to amplify the difference between
```

(continues on next page)

(continued from previous page)

```
# simple time delta get_total_time() and the get_total_execution_time().
time.sleep(1.0)
instr.reset()

print('\nCommunication time spent: ' + str(instr.get_total_execution_time()))
print('Program time spent:       ' + str(instr.get_total_time()))

instr.reset_time_statistics()
print('\nWe can reset the time stats to start from 0 again:\n')

# Also, the next log entry will have the start time set to 00:00:00.000
instr.logger.set_time_offset_zero_on_first_entry()

# Again, pause for 500 milliseconds to see
# the difference between the get_total_time() and the get_total_execution_time().
time.sleep(0.5)
instr.check_status()

# Close the session
instr.close()

print('\nCommunication time spent: ' + str(instr.get_total_execution_time()))
print('Program time spent:       ' + str(instr.get_total_time()))
```

Console output:

```
00:00:00.000  TCPIP::192.168.1.101::hislip0   137.704 ms  Session init: Device
↪'TCPIP::192.168.1.101::hislip0' IDN: Rohde&Schwarz,SMA100B,1419.8888K02/0,5.30.132.68
00:00:00.137  TCPIP::192.168.1.101::hislip0     2.002 ms  Status check: OK

Entries above come from the constructor.

00:00:00.139  TCPIP::192.168.1.101::hislip0     2.922 ms  Query: *IDN? Rohde&Schwarz,
↪SMA100B,1419.8888K02/0,5.30.132.68
00:00:00.142  TCPIP::192.168.1.101::hislip0     1.510 ms  Status check: OK
00:00:01.145  TCPIP::192.168.1.101::hislip0    32.979 ms  Write: *RST
00:00:01.178  TCPIP::192.168.1.101::hislip0  1879.281 ms  Status check: OK
00:00:03.057  TCPIP::192.168.1.101::hislip0   129.349 ms  Query OPC: 1
00:00:03.186  TCPIP::192.168.1.101::hislip0     5.432 ms  Clear status: OK
00:00:03.192  TCPIP::192.168.1.101::hislip0     1.982 ms  Status check: OK

Communication time spent: 0:00:02.191159
Program time spent:       0:00:03.194408

We can reset the time stats to start from 0 again:

00:00:00.000  TCPIP::192.168.1.101::hislip0    33.615 ms  Status check: OK
00:00:00.033  TCPIP::192.168.1.101::hislip0     1.952 ms  Close: Closing session

Communication time spent: 0:00:00.001952
Program time spent:       0:00:00.536126
```

18.4 Integration with Python's logging module

Commonly used Python's logging can be used with RsInstrument too:

```
"""
Example of logging to a python standard logger object.
"""

import logging

from RsInstrument import *

RsInstrument.assert_minimum_version('1.102.0')


class LoggerStream:
    """Class to wrap the python's logging into a stream interface."""

    @staticmethod
    def write(log_entry: str) -> None:
        """Method called by the RsInstrument to add the log_entry.
        Use it to do your custom operation, in our case calling python's logging function.
        """
        logging.info('RsInstrument: ' + log_entry.rstrip())

    def flush(self) -> None:
        """Do the operations at the end. In our case, we do nothing."""
        pass


# Setting of the SMW
smw = RsInstrument('TCPIP::10.99.2.10::hislip0', options='LoggingMode=On, LoggingName=SMW')

# Create a logger stream object
target = LoggerStream()
logging.getLogger().setLevel(logging.INFO)

# Adjust the log string to not show the start time
smw.logger.set_format_string('PAD_LEFT25(%DEVICE_NAME%) PAD_LEFT12(%DURATION%)  %LOG_STRING_INFO%: %LOG_STRING%')
smw.logger.set_logging_target(target)  # Log to my target

smw.logger.info_raw("> Custom log from SMW session")
smw.reset()

# Close the sessions
smw.close()
```

18.5 Logging from multiple sessions

We hope you are a happy Rohde & Schwarz customer, and hence you use more than one of our instruments. In such case, you probably want to log from all the instruments into a single target (file). Therefore, you open one log file for writing (or appending) and the set is as the logging target for all your sessions:

```
"""
Example of logging to a file shared by multiple sessions.
"""

from RsInstrument import *

RsInstrument.assert_minimum_version('1.102.0')

# Log file common for all the instruments,
# previous content is discarded.
file = open(r'c:\temp\my_file.txt', 'w')

# Setting of the SMW
smw = RsInstrument('TCPIP::192.168.1.101::INSTR', options='LoggingMode=On,␣
↪LoggingName=SMW')
smw.logger.set_logging_target(file, console_log=True)  # Log to file and the console

# Setting of the SMCV
smcv = RsInstrument('TCPIP::192.168.1.102::INSTR', options='LoggingMode=On,␣
↪LoggingName=SMCV')
smcv.logger.set_logging_target(file, console_log=True)  # Log to file and the console

smw.logger.info_raw("> Custom log from SMW session")
smw.reset()
smcv.logger.info_raw("> Custom log from SMCV session")
idn = smcv.query('*IDN?')

# Close the sessions
smw.close()
smcv.close()

# Close the log file
file.close()
```

Console output:

```
11:43:42.657            SMW    10.712 ms  Session init: Device 'TCPIP::192.168.1.
↪101::INSTR' IDN: Rohde&Schwarz,SMW200A,1412.0000K02/0,4.70.026 beta
11:43:42.668            SMW     2.928 ms  Status check: OK
11:43:42.686           SMCV     1.952 ms  Session init: Device 'TCPIP::192.168.1.
↪102::INSTR' IDN: Rohde&Schwarz,SMCV100B,1432.7000K02/0,4.70.060.41 beta
11:43:42.688           SMCV     1.981 ms  Status check: OK
> Custom log from SMW session
11:43:42.690            SMW     0.973 ms  Write: *RST
11:43:42.690            SMW  1874.658 ms  Status check: OK
11:43:44.565            SMW     0.976 ms  Query OPC: 1
11:43:44.566            SMW     1.952 ms  Clear status: OK
```

(continues on next page)

(continued from previous page)

```
11:43:44.568            SMW     2.928 ms  Status check: OK
> Custom log from SMCV session
11:43:44.571           SMCV     0.975 ms  Query: *IDN? Rohde&Schwarz,SMCV100B,1432.
↪7000K02/0,4.70.060.41 beta
11:43:44.571           SMCV     1.951 ms  Status check: OK
11:43:44.573            SMW     0.977 ms  Close: Closing session
11:43:44.574           SMCV     0.976 ms  Close: Closing session
```

Tip

To make the log more compact, you can skip all the lines with `Status check: OK`:

```
smw.logger.log_status_check_ok = False
```

18.6 Logging to UDP

For logging to a UDP port in addition to other log targets, use one of the lines:

```
smw.logger.log_to_udp = True
smw.logger.log_to_console_and_udp = True
```

You can select the UDP port to log to, the default is 49200:

```
smw.logger.udp_port = 49200
```

18.7 Logging from all instances

In Python everything is an object. Even class definition is an object that can have attributes. We can habe logging target as a class variable (class attribute). The interesting effect of a class variable is, that it has immediate effect on all its instances. Let us rewrite the example above for multiple sessions and use the class variable not only for the log target, but also a relative timestamp, which gives us the log output starting from relative time **00:00:00:000**. The created log file will have the same name as the script, but with the extension .ptc (dedicated to those who still remember R&S Forum :-)

```
"""
Example of logging to a file shared by multiple sessions.
The log file and the reference timestamp is set to the RsInstrument class variable,
which makes it available to all the instances immediately.
Each instance must set the LogToGlobalTarget=True in the constructor,
or later io.logger.set_logging_target_global().
"""

from RsInstrument import *
import os
from pathlib import Path
from datetime import datetime

RsInstrument.assert_minimum_version('1.102.0')

```

(continues on next page)

(continued from previous page)

```
# Log file common for all the RsInstrument instances, saved in the same folder as this script,
# with the same name as this script, just with the suffix .ptc
# The previous file content is discarded.
log_file = open(Path(os.path.realpath(__file__)).stem + ".ptc", 'w')
RsInstrument.set_global_logging_target(log_file)

# If you do now care about the absolute times,
# here you can set relative timestamp of the first incoming entry.
RsInstrument.set_global_logging_relative_time_of_first_entry()

# Setting of the SMW: log to the global target and to the console
smw = RsInstrument(
    resource_name='TCPIP::10.102.12.13::hislip0',
    options=f'LoggingMode=On, LoggingToConsole=True, LoggingName=SMW, LogToGlobalTarget=On')

# Setting of the SMCV: log to the global target and to the console
smcv = RsInstrument(
    resource_name='TCPIP::10.102.12.14::hislip0',
    options='LoggingMode=On, LoggingToConsole=True, LoggingName=SMCV, LogToGlobalTarget=On')

smw.logger.info_raw("> Custom log entry from SMW session")
smw.reset()
smcv.logger.info_raw("> Custom log entry from SMCV session")
idn = smcv.query('*IDN?')
# Close the sessions
smw.close()
smcv.close()
# Show how much time each instrument needed for its operations.
smw.logger.info_raw("> SMW execution time: " + str(smw.get_total_execution_time()))
smcv.logger.info_raw("> SMCV execution time: " + str(smcv.get_total_execution_time()))

# Close the log file
log_file.close()
```

Console output and the file content:

```
00:00:00.000                              SMW   117.739 ms  Session init: Device 'TCPIP::192.168.1.101::hislip0' IDN: Rohde&Schwarz,SMW200A,1412.0000K02/0,5.30.305.44
00:00:00.119                              SMW     0.982 ms  Status check: OK
00:00:00.120                             SMCV    54.835 ms  Session init: Device 'TCPIP::192.168.1.102::hislip0' IDN: Rohde&Schwarz,SMCV100B,1432.7000K02/0,5.30.175.95
00:00:00.175                             SMCV     0.984 ms  Status check: OK
> Custom log entry from SMW session
00:00:00.176                              SMW     0.000 ms  Write: *RST
00:00:00.176                              SMW  1633.359 ms  Status check: OK
00:00:01.809                              SMW     7.391 ms  Query OPC: 1
00:00:01.816                              SMW    10.337 ms  Clear status: OK
00:00:01.827                              SMW     1.952 ms  Status check: OK
> Custom log entry from SMCV session
```

(continues on next page)

(continued from previous page)

```
00:00:01.829                              SMCV    42.493 ms  Query: *IDN? Rohde&Schwarz,
↪SMCV100B,1432.7000K02/0,5.30.175.95
00:00:01.871                              SMCV     1.953 ms  Status check: OK
00:00:01.873                               SMW    22.739 ms  Close: Closing session
00:00:01.896                              SMCV    23.160 ms  Close: Closing session
> SMW execution time: 0:00:01.793517
> SMCV execution time: 0:00:00.122441
```

For the completion, here are all the global time functions:

```
RsInstrument.set_global_logging_relative_timestamp(timestamp: datetime)
RsInstrument.get_global_logging_relative_timestamp() -> datetime
RsInstrument.set_global_logging_relative_timestamp_now()
RsInstrument.clear_global_logging_relative_timestamp()
```

and the session-specific time and statistic methods:

```
smw.logger.set_relative_timestamp(timestamp: datetime)
smw.logger.set_relative_timestamp_now()
smw.logger.get_relative_timestamp() -> datetime
smw.logger.clear_relative_timestamp()

smw.get_total_execution_time() -> timedelta
smw.get_total_time() -> timedelta
smw.get_total_time_startpoint() -> datetime
smw.reset_time_statistics()
```

18.8 Logging only errors

Another neat feature is errors-only logging. To make this mode useful for troubleshooting, you also want to see the circumstances which lead to the errors. Each RsInstrument elementary operation, for example, `write()`, can generate a group of log entries - let us call them **Segment**. In the logging mode `Errors`, a whole segment is logged only if at least one entry of the segment is an error.

The script below demonstrates this feature. We deliberately misspelled a SCPI command *CLS, which leads to instrument status error:

```
"""
Logging example to the console with only errors logged.
"""

from RsInstrument import *

RsInstrument.assert_minimum_version('1.102.0')
instr = RsInstrument('TCPIP::192.168.1.101::INSTR', options='LoggingMode=Errors')

# Switch ON logging to the console.
instr.logger.log_to_console = True

# Reset will not be logged, since no error occurred there
instr.reset()

```

(continues on next page)

(continued from previous page)

```
# Now a misspelled command.
instr.write('*CLaS')

# A good command again, no logging here
idn = instr.query('*IDN?')

# Close the session
instr.close()
```

Console output:

```
12:11:02.879 TCPIP::192.168.1.101::INSTR     0.976 ms  Write: *CLaS
12:11:02.879 TCPIP::192.168.1.101::INSTR     6.833 ms  Status check: StatusException:
                                                       Instrument error detected: Undefined header;
↪*CLaS
```

Notice the following:

- Although the operation **Write: *CLaS** finished without an error, it is still logged, because it provides the context for the actual error which occurred during the status checking right after.
- No other log entries are present, including the session initialization and close, because they ran error-free.

18.9 Setting the logging format

You can adjust the logging to your liking by setting the format string. The default format string:

```
PAD_LEFT12(%START_TIME%) PAD_LEFT25(%DEVICE_NAME%) PAD_LEFT12(%DURATION%)
%LOG_STRING_INFO%:  %LOG_STRING%
```

Here's an example for you minimalists, who only want to see the start, duration, and the SCPI command:

```
"""
Logging only the SCPI commands to the console.
"""

from RsInstrument import *

RsInstrument.assert_minimum_version('1.102.0')
instr = RsInstrument('TCPIP::10.102.52.53::hislip0')

# Switch ON logging to the console.
instr.logger.log_to_console = True
instr.logger.set_format_string('PAD_LEFT12(%START_TIME%) PAD_LEFT12(%DURATION%) %SCPI_
↪COMMAND%')
instr.logger.start()
instr.reset()

# Close the session
instr.close()
```

Console output:

```
09:31:54.146    40.991 ms *RST
09:31:54.146  1939.319 ms *STB?
09:31:56.086    81.984 ms *OPC?
09:31:56.168   124.930 ms *CLS
09:31:56.293    82.015 ms *STB?
09:31:56.375   144.447 ms @CLOSE_SESSION
```

You can also initiate the logging and change its format in the constructor options string. The console output logs everything including the session intialization commands:

```
"""
Logging only the SCPI commands to the console.
"""

from RsInstrument import *

RsInstrument.assert_minimum_version('1.102.0')
instr = RsInstrument('TCPIP::10.102.52.53::hislip0',
             options='LoggingMode=On, '
                     'LoggingToConsole=True, '
                     'LoggingFormat = "PAD_LEFT12(%START_TIME%) PAD_LEFT12(%DURATION%)
↪%SCPI_COMMAND%"')
instr.reset()

# Close the session
instr.close()
```

Console output:

```
09:36:47.983  1210.274 ms @INIT_SESSION
09:36:49.193    81.984 ms *STB?
09:36:49.275    40.988 ms *RST
09:36:49.275  1929.529 ms *STB?
09:36:51.205    82.990 ms *OPC?
09:36:51.288   124.899 ms *CLS
09:36:51.413    81.987 ms *STB?
09:36:51.495   144.480 ms @CLOSE_SESSION
```

Another possible customization is keeping the %LOG_STRING_INFO%, but replacing its content with your own strings. Let us make more compact output that way:

```
"""
Logging with customized log info string to the console.
"""

from RsInstrument import *


RsInstrument.assert_minimum_version('1.102.0')
instr = RsInstrument('TCPIP::10.102.52.53::hislip0')

# Switch ON logging to the console.
instr.logger.log_to_console = True
```

(continues on next page)

(continued from previous page)

```
instr.logger.set_format_string('PAD_LEFT12(%START_TIME%) PAD_LEFT12(%DURATION%) PAD_LEFT9(%LOG_STRING_INFO%):  %SCPI_COMMAND%')

# We will replace the 'Write' and 'Query' log string infos with only 'W' and 'Q' - full match and replace:
instr.logger.log_info_replacer.put_full_replacer_item(match = 'Write', replace = 'W')
instr.logger.log_info_replacer.put_full_replacer_item(match = 'Query', replace = 'Q')
instr.logger.log_info_replacer.put_full_replacer_item(match = 'Clear status', replace = 'Cls')
instr.logger.log_info_replacer.put_full_replacer_item(match = 'Status check', replace = 'Q_err')

# If the full match does not fit your needs, use the regex search and replace:
# This regex will replace the 'Query OPC' with 'Q_OPC' and 'Query integer' with 'Q_integer'
instr.logger.log_info_replacer.put_regex_sr_replacer_item(r'^Query (.+)$', r'Q_\1')

instr.logger.start()
instr.reset()
instr.query('*IDN?')
instr.query_int('*STB?')

# Close the session
instr.close()
```

Console output:

```
12:19:37.025     0.975 ms         W:  *RST
12:19:37.025  1879.123 ms     Q_err:  *STB?
12:19:38.905     0.976 ms     Q_OPC:  *OPC?
12:19:38.906     1.951 ms       Cls:  *CLS
12:19:38.908     0.000 ms     Q_err:  *STB?
12:19:38.908     1.007 ms         Q:  *IDN?
12:19:38.908     1.952 ms     Q_err:  *STB?
12:19:38.910     1.002 ms Q_integer:  *STB?
12:19:38.910     1.984 ms     Q_err:  *STB?
12:19:38.912    23.392 ms     Close:  @CLOSE_SESSION
```

CHAPTER

NINETEEN

MCP SERVER

The module also provides a simple MCP server that allows remote control of R&S instruments without the need to install any VISA library on the agent side.

> **Note**
>
> Please be aware that you need Python >= 3.10 to run the MCP server.

```
RsInstrument-mcp --host localhost --port 8000 --transport streamable-http
```

Extend the built-in tools with your own using this blueprint:

```
from RsInstrument import RsInstrument
from RsInstrument.mcp import run, safe_tool


@safe_tool  # Decorator to catch and log exceptions and return them as error messages to the agent
def instrument_fancy_function(resource: str, opc_timeout: int = 5000) -> str:
    """My fancy function for RsInstrument MCP.

    Args:
        resource: The VISA resource string of the instrument.
        opc_timeout: Timeout in milliseconds for the operation complete (OPC) query.
            Default is 5000 ms.

    Returns:
        The response from the instrument.
    """
    with RsInstrument(resource) as inst:
        inst.opc_timeout = opc_timeout
        # your logic goes here
        return "RsInstrument is awesome."


if __name__ == "__main__":
    run(
        host="localhost",
        port=8000,
        transport="streamable-http",
```

(continues on next page)

(continued from previous page)

```
        tools=[
            (
                "Instrument-Fancy-SCPI",  # Tool name
                "This is an awesome function",  # Tool description
                instrument_fancy_function,  # Tool function
            )
        ],
    )
```

After starting the server, you can access the tools at http://localhost:8000/mcp.

CHAPTER

TWENTY

RSINSTRUMENT PACKAGE

20.1 Modules

20.2 RsInstrument.RsInstrument

Root class for remote-controlling instrument with SCPI commands.

class RsInstrument(*resource_name: str*, *id_query: bool = True*, *reset: bool = False*, *options: str = None*, *direct_session: object = None*)

Bases: `object`

Root class for remote-controlling instrument with SCPI commands.

Initializes new RsInstrument session.

Parameters

- **resource_name** – VISA resource name, e.g. 'TCPIP::192.168.2.1::INSTR'
- **id_query** – if True, the instrument's model name is verified against the models supported by the driver and eventually throws an exception
- **reset** – Resets the instrument (sends *RST) command and clears its status syb-system
- **direct_session** – Another driver object or pyVisa object to reuse the session instead of opening a new session
- **options** – string tokens alternating the driver settings. More tokens are separated by comma.

Parameter options tokens examples:

- `Simulate=True` - starts the session in simulation mode. Default: `False`
- `SelectVisa=socketio` - uses no VISA implementation for socket connections - you do not need any VISA-C installation
- `SelectVisa=rs` - forces usage of RohdeSchwarz Visa
- `SelectVisa=ni` - forces usage of National Instruments Visa
- `Profile = HM8123` - setting profile fitting the specific non-standard instruments. Available values: HM8123, CMQ, ATS, Minimal. Default: `none`
- `OpenTimeout=5000` - sets timeout used at the session opening. This timeout is only used in waiting for a locked session to be freed. Default: `2000ms`
- `ExclusiveLock=True` - opens the session with exclusive lock on the VISA level. Default: `False`

- `QueryInstrumentStatus = False` - same as `driver.utilities.instrument_status_checking = False`. Default: `True`
- `WriteDelay = 20, ReadDelay = 5` - introduces delay of 20ms before each write and 5ms before each read. Default: `0ms` for both
- `TerminationCharacter = "\r"` - sets the termination character for reading. Default: `\n` (LineFeed or LF)
- `AssureWriteWithTermChar = True` - makes sure each command/query is terminated with termination character. Default: Interface dependent
- `AddTermCharToWriteBinBlock = True` - adds one additional LF to the end of the binary data (some instruments require that). Default: `False`
- `DataChunkSize = 10E3` - maximum size of one write/read segment. If transferred data is bigger, it is split to more segments. Default: `1E7` bytes
- `OpcTimeout = 10000` - same as driver.utilities.opc_timeout = 10000. Default: `30000ms`
- `VisaTimeout = 5000` - same as driver.utilities.visa_timeout = 5000. Default: `10000ms`
- `ViClearExeMode = Disabled` - viClear() execution mode. Default: `execute_on_all`
- `OpcQueryAfterWrite = True` - same as driver.utilities.opc_query_after_write = True. Default: `False`
- `OpcWaitMode = OpcQuery` - mode for all the opc-synchronised write/reads. Other modes: StbPolling, StbPollingSlow, StbPollingSuperSlow. Default: `StbPolling`
- `StbInErrorCheck = False` - if true, the driver checks errors with *STB? If false, it uses SYST:ERR?. Default: `True`
- `SkipStatusSystemSettings = False` - some instruments do not support full status system commands. In such case, set this value to True. Default: `False`
- `SkipClearStatus = True` - set to True for instruments that do not support *CLS command. Default: `False`
- `DisableOpcQuery = True` - set to True for instruments that do not support *OPC? query. Default: `False`
- `EachCmdAsQuery = True`, set to True, for instruments that always return answer. Default: `false`
- `CmdIdn = ID?` - defines which SCPI command to use for identification query. Use '<none>' string to skip identification query at the init. Default: `*IDN?`
- `CmdReset = RT` - defines which SCPI command to use for reset. Default: `*RST`
- `VxiCapable = false` - you can force a session to a VXI-incapable. Default: <interface-dependent>
- `Encoding = utf-8` - setting of encoding for strings into bytes and vice versa. Default: `charmap`
- `OpcSyncQueryMechanism = AlsoCheckMav` - setting of mechanism for OPC-synchronised queries. Default: `OnlyCheckMavErrQueue`
- `FirstCmds = *CLS` - first command(s) to sent after init. Separated more commands/queries with ';;'. Default: ````
- `EachCmdPrefix = lf` - this prefix is added to the beginning of each command sent to the instrument. Default: ````
- `EachCmdSuffix = cr` - this suffix is added to the end of each command sent to the instrument. Default: ````

- `StripStringTrailingWhitespaces = True` - use it to strip white spaces from string query responses. Default: `False`
- `LoggingMode = On` - sets the logging status right from the start. Possible values: On | Off | Error. Default: `Off`
- `LoggingName = 'MyDevice'` - sets the name to represent the session in the log entries. Default: `<resource_name>`
- `LoggingFormat = 'PAD_LEFT12(%START_TIME%) PAD_LEFT25(%DEVICE_NAME%) PAD_LEFT12(%DURATION%) %SCPI_COMMAND%'` - sets the format of the log entries. Default: `PAD_LEFT12(%START_TIME%) PAD_LEFT25(%DEVICE_NAME%) PAD_LEFT12(%DURATION%) %LOG_STRING_INFO%: %LOG_STRING%`
- `LogToGlobalTarget = True` - sets the logging target to the class-property previously set with RsInstrument.set_global_logging_target() Default: `False`
- `LoggingToConsole = True` - immediately starts logging to the console. Default: False
- `LoggingToUdp = True` - immediately starts logging to the UDP port. Default: False
- `LoggingUdpPort = 49200` - UDP port to log to. Default: 49200
- `LoggingRelativeTimeOfFirstEntry = True` - Logging starts with relative time set to the first log entry, which causes the first start time to be '00:00:00.000'. Default: False

add_instr_option(*option: str*) → None

Adds new option if not already existing.

static assert_minimum_version(*min_version: str*) → None

Asserts that the driver version fulfills the minimum required version you have entered. This way you make sure your installed driver is of the entered version or newer.

assign_lock(*lock: RLock*) → None

Assigns the provided thread lock.

property bin_float_numbers_format: *BinFloatFormat*

Sets / returns format of float numbers when transferred as binary data

property bin_int_numbers_format: *BinIntFormat*

Sets / returns format of integer numbers when transferred as binary data

check_status() → None

Throws InstrumentStatusException in case of an error in the instrument's error queue. The status checking is performed always, independent of the property 'instrument_status_checking'. Also, the property ScpiLogger.log_status_check_ok is ignored, and the Status check is always logged.

classmethod clear_global_logging_relative_timestamp() → None

Clears the global relative timestamp. After this, all the instances using the global relative timestamp continue logging with the absolute timestamps.

clear_lock() → None

Clears the existing thread lock, making the current session thread-independent from others that might share the current thread lock.

clear_status() → None

Clears instrument's status system, the session's I/O buffers and the instrument's error queue

close() → None

Closes the active RsInstrument session

property data_chunk_size: int

Returns max chunk size of one data block.

property driver_version: str

Returns the RsInstrument package version

property encoding: str

Returns string<=>bytes encoding of the session.

property events: *Events*

Interface for event handlers, see *here*

file_exists(*instr_file: str*) → bool

Returns true, if the instrument file exist.

classmethod from_existing_session(*session: object*, *options: str = None*) → *RsInstrument*

Creates a new RsInstrument object with the entered 'session' reused. :param session: can be another driver or a direct pyvisa session. :param options: string tokens alternating the driver settings. More tokens are separated by comma.

property full_instrument_model_name: str

Returns the current instrument's full name e.g. 'FSW26'

classmethod get_driver_version() → str

Returns the RsInstrument package version

get_file_size(*instr_file: str*) → int | None

Return size of the instrument file, or None if the file does not exist.

classmethod get_global_logging_relative_timestamp() → datetime | None

Returns global common relative timestamp for log entries.

classmethod get_global_logging_target()

Returns global common target stream.

get_last_sent_cmd() → str

Returns the last commands sent to the instrument. Only works in simulation mode.

get_lock() → RLock

Returns the thread lock for the current session.

By default:

- If you create a new RsInstrument instance with new VISA session, the session gets a new thread lock. You can assign it to another RsInstrument sessions in order to share one physical instrument with a multi-thread access.
- If you create a new RsInstrument from an existing session, the thread lock is shared automatically making both instances multi-thread safe.

You can always assign new thread lock by calling `driver.utilities.assign_lock()`

get_session_handle()

Returns the underlying pyvisa session

get_total_execution_time() → timedelta

Returns total time spent by the library on communicating with the instrument. This time is always shorter than get_total_time(), since it does not include gaps between the communication. You can reset this counter with reset_time_statistics().

get_total_time() → timedelta

Returns delta time spent by the library between the get_total_time_startpoint() and now. This time is always longer than get_total_execution_time(), since it also includes all other activities besides the communication. You can set the total time startpoint to now with reset_time_statistics().

get_total_time_startpoint() → datetime

Returns time from which the execution started. This is the value that the get_total_time() calculates as its reference. Calling the reset_time_statistics() sets this time to now.

go_to_local(*mixed_mode: bool = True*) → None

Puts the instrument into local state. By default, the method uses a mechanism to keep the instrument in a mixed mode: remote and local. That means, you can remote-control your instrument, and at the same time it still allows manual control. Set the mixed_mode to False, if you want your instrument to go to remote mode as soon as it receives the first remote command.

go_to_remote() → None

Puts the instrument into remote state.

has_instr_option(*options: str | List[str]*) → bool

Returns true, if the entered options (case-insensitive) matches at least one of the installed options (or-logic). You can enter either a string with one option, or more options '/'-separated, or more options as a list of strings. If K0 is present, all the K-options are reported as present. B-options are not affected by K0. Example 1: options='k23' returns true, if the instrument has the option 'K23'. Example 2: options='k23 / K23e' returns true, if the instrument has either the option 'K23' or the option 'K23E'. Example 3: options=['k11','K22'] returns true, if the instrument has either the option 'K11' or the option 'K22'.

has_instr_option_k0() → bool

Returns true, if the instrument has K0 installed.

has_instr_option_regex(*re_options: str | List[str]*) → bool

Returns true, if the entered regex string (case-insensitive) matches at least one of the installed options. The match must be complete, not just partial (search). You can enter either a string with one option, or more options '/'-separated, or more options as a list of strings. Example 1: re_options='k10.' returns true, if the instrument contains any option 'K100' … up to 'K109' . Example 2: re_options='k10. / k20.*' returns true, if the instrument contains any of the options 'K10x' or 'K20xxx'. Example 3: re_options=['k10.', 'k20.*'] returns true, if the instrument contains any options 'K10x' or 'K20xxx'.

property idn_string: str

Returns instrument's identification string - the response on the SCPI command *IDN?

instr_err_suppressor(*visa_tout_ms: int = 0*, *suppress_only_codes: int | List[int] = None*) → InstrErrorSuppressor

Returns Context Manager that suppresses the instrument errors. Other exceptions types are still raised. On entering the context, this class clears all the instrument status errors. :param visa_tout_ms: VISA Timeout in milliseconds, that is set for this context. Afterward, it is changed back. Default value: do-not-change. :param suppress_only_codes: You can enter a code or list of codes for errors to be suppressed. Other errors will be reported. Example: If you enter -113 here, only the 'Undefined Header' error will be suppressed. Default value: suppress-all-errors.

property instrument_firmware_version: str

Returns instrument's firmware version

property instrument_model_name: str

Returns the current instrument's family name e.g. 'FSW'

property instrument_options: List[str]

Returns all the instrument options. The options are sorted in the ascending order starting with K-options and continuing with B-options

property instrument_serial_number: str

Returns instrument's serial_number

property instrument_status_checking: bool

Sets / returns Instrument Status Checking. When True (default is True), all the driver methods and properties are sending "SYSTem:ERRor?" at the end to immediately react on error that might have occurred. We recommend keeping the state checking ON all the time. Switch it OFF only in rare cases when you require maximum speed. The default state after initializing the session is ON.

is_connection_active() → bool

Returns true, if the VISA connection is active and the communication with the instrument still works. WARNING!!! this method queries the session's VISA Timeout and additionally, queries the *IDN? from the instrument, hence affects the performance of your application when used regularly.

static list_resources(*expression: str = '?*::INSTR'*, *visa_select: str = None*) → List[str]

Finds all the resources defined by the expression.

- '?*' - matches all the available instruments
- 'USB::?*' - matches all the USB instruments
- 'TCPIP::192?*' - matches all the LAN instruments with the IP address starting with 192

Parameters

- **expression** – see the examples in the function
- **visa_select** – optional parameter selecting a specific VISA. Examples: '@ivi', '@rs'

lock_resource(*timeout: int*, *requested_key: str | bytes = None*) → bytes | str

Locks the instrument to prevent it from communicating with other clients.

property logger: *ScpiLogger*

Scpi Logger interface, see *here*

property manufacturer: str

Returns manufacturer of the instrument

property opc_query_after_write: bool

Sets / returns Instrument *OPC? query sending after each command write. When True, (default is False) the driver sends *OPC? every time a write command is performed. Use this if you want to make sure your sequence is performed command-after-command.

property opc_sync_query_mechanism: *OpcSyncQueryMechanism*

Returns the current setting of the OPC-Sync query mechanism.

property opc_timeout: int

Sets / returns timeout in milliseconds for all the operations that use OPC synchronization.

process_all_commands() → None

SCPI command: *WAI Stops further commands processing until all commands sent before *WAI have been executed.

query(*query: str*) → str

Sends the string query to the instrument and returns the response as string. The response is trimmed of any trailing LF characters and has no length limit. This method is an alias to the query_str() method.

query_all_errors() → List[str] | None

Queries and clears all the errors from the instrument's error queue. The method returns list of strings as error messages. If no error is detected, the return value is None. The process is: querying 'SYSTem:ERRor?' in a loop until the error queue is empty. If you want to include the error codes, call the query_all_errors_with_codes()

query_all_errors_with_codes() → List[Tuple[int, str]] | None

Queries and clears all the errors from the instrument's error queue. The method returns list of tuples (code: int, message: str). If no error is detected, the return value is None. The process is: querying 'SYSTem:ERRor?' in a loop until the error queue is empty.

query_bin_block(*query: str*) → bytes

Queries binary data block to bytes. Throws an exception if the returned data was not a binary data. Returns data:bytes

query_bin_block_to_file(*query: str*, *file_path: str*, *append: bool = False*) → None

Queries binary data block to the provided file. If append is False, any existing file content is discarded. If append is True, the new content is added to the end of the existing file, or if the file does not exit, it is created. Throws an exception if the returned data was not a binary data. Example for transferring a file from Instrument -> PC: query = f"MMEM:DATA? '{INSTR_FILE_PATH}'".

Alternatively, use the dedicated methods for this purpose:

- `send_file_from_pc_to_instrument()`
- `read_file_from_instrument_to_pc()`

query_bin_block_to_file_with_opc(*query: str*, *file_path: str*, *append: bool = False*, *timeout: int = None*) → None

Sends a OPC-synced query and writes the returned data to the provided file. If append is False, any existing file content is discarded. If append is True, the new content is added to the end of the existing file, or if the file does not exit, it is created. Throws an exception if the returned data was not a binary data.

query_bin_block_with_opc(*query: str*, *timeout: int = None*) → bytes

Sends a OPC-synced query and returns binary data block to bytes. If you do not provide timeout, the method uses current opc_timeout.

query_bin_or_ascii_float_list(*query: str*) → List[float]

Queries a list of floating-point numbers that can be returned as ASCII or binary format.

- For ASCII format, the list numbers are decoded as comma-separated values.
- For Binary Format, the numbers are decoded based on the property BinFloatFormat, usually float 32-bit (FORM REAL,32).

query_bin_or_ascii_float_list_with_opc(*query: str*, *timeout: int = None*) → List[float]

Sends a OPC-synced query and reads a list of floating-point numbers that can be returned as ASCII or binary format.

- For ASCII format, the list numbers are decoded as comma-separated values.
- For Binary Format, the numbers are decoded based on the property BinFloatFormat, usually float 32-bit (FORM REAL,32).

If you do not provide timeout, the method uses current opc_timeout.

query_bin_or_ascii_int_list(*query: str*) → List[int]

Queries a list of floating-point numbers that can be returned as ASCII or binary format.

- For ASCII format, the list numbers are decoded as comma-separated values.
- For Binary Format, the numbers are decoded based on the property BinFloatFormat, usually float 32-bit (FORM REAL,32).

query_bin_or_ascii_int_list_with_opc(*query: str*, *timeout: int = None*) → List[int]

Sends a OPC-synced query and reads a list of floating-point numbers that can be returned as ASCII or binary format.

- For ASCII format, the list numbers are decoded as comma-separated values.
- For Binary Format, the numbers are decoded based on the property BinFloatFormat, usually float 32-bit (FORM REAL,32).

If you do not provide timeout, the method uses current opc_timeout.

query_bool(*query: str*) → bool

Sends the query to the instrument and returns the response as boolean.

query_bool_list(*query: str*) → List[bool]

Sends the string query to the instrument and returns the response as List of booleans, where the delimiter is comma (‘,’). Blank or empty response is returned as an empty list.

query_bool_list_with_opc(*query: str*, *timeout: int = None*) → List[bool]

Sends a OPC-synced query and reads response from the instrument as csv-list of booleans. If you do not provide timeout, the method uses current opc_timeout. Blank or empty response is returned as an empty list.

query_bool_with_opc(*query: str*, *timeout: int = None*) → bool

Sends the opc-synced query to the instrument and returns the response as boolean. If you do not provide timeout, the method uses current opc_timeout.

query_float(*query: str*) → float

Sends the query to the instrument and returns the response as float.

query_float_with_opc(*query: str*, *timeout: int = None*) → float

Sends the opc-synced query to the instrument and returns the response as float. If you do not provide timeout, the method uses current opc_timeout.

query_int(*query: str*) → int

Sends the query to the instrument and returns the response as integer.

query_int_with_opc(*query: str*, *timeout: int = None*) → int

Sends the opc-synced query to the instrument and returns the response as integer. If you do not provide timeout, the method uses current opc_timeout.

query_opc(*timeout: int = 0*) → int

SCPI command: *OPC? Queries the instrument’s OPC bit and hence it waits until the instrument reports operation complete. If you define timeout > 0, the VISA timeout is set to that value just for this method call.

query_str(*query: str*) → str

Sends the string query to the instrument and returns the response as string. The response is trimmed of any trailing LF characters and has no length limit. This method is an alias to the query() method.

query_str_list(*query: str*, *remove_blank_response: bool = False*) → List[str]

Sends the string query to the instrument and returns the response as List of strings, where the delimiter is comma (‘,’). Each element of the list is trimmed for leading and trailing quotes.

Meaning of the ‘remove_blank_response’:

- False(default): whitespaces-only response is returned as a list with one empty element [‘’].
- True: whitespaces-only response is returned as an empty list [].

query_str_list_with_opc(*query: str*, *timeout: int = None*, *remove_blank_response: bool = False*) → List[str]

Sends a OPC-synced query and reads response from the instrument as csv-list.
If you do not provide timeout, the method uses current opc_timeout.

Meaning of the ‘remove_blank_response’:

- False(default): whitespaces-only response is returned as a list with one empty element [‘’].
- True: whitespaces-only response is returned as an empty list [].

query_str_stripped(*query: str*) → str

Sends the string query to the instrument and returns the response as string stripped of the trailing LF and leading/trailing single/double quotes. The stripping of the leading/trailing quotes is blocked, if the string contains the quotes in the middle. This method is an alias to the query_stripped() method.

query_str_with_opc(*query: str*, *timeout: int = None*) → str

Sends the opc-synced query to the instrument and returns the response as string. The response is trimmed of any trailing LF characters and has no length limit. If you do not provide timeout, the method uses current opc_timeout.

query_stripped(*query: str*) → str

Sends the string query to the instrument and returns the response as string stripped of the trailing LF and leading/trailing single/double quotes. The stripping of the leading/trailing quotes is blocked, if the string contains the quotes in the middle.

query_with_opc(*query: str*, *timeout: int = None*) → str

This method is an alias to the write_str_with_opc(). Sends the opc-synced query to the instrument and returns the response as string. The response is trimmed of any trailing LF characters and has no length limit. If you do not provide timeout, the method uses current opc_timeout.

read_file_from_instrument_to_pc(*source_instr_file: str*, *target_pc_file: str*, *append_to_pc_file: bool = False*) → None

SCPI Command: MMEM:DATA?

Reads file from instrument to the PC.

Set the `append_to_pc_file` to True if you want to append the read content to the end of the existing PC file.

reconnect(*force_close: bool = False*) → bool

If the connection is not active, the method tries to reconnect to the device. If the connection is active, and force_close is False, the method does nothing. If the connection is active, and force_close is True, the method closes, and opens the session again. Returns True, if the reconnection has been performed.

remove_instr_option(*option: str*) → None

Removes the option if exists.

reset(*timeout: int = 0*) → None

SCPI command: *RST Sends *RST command + calls the clear_status(). If you define timeout > 0, the VISA timeout is set to that value just for this method call.

reset_time_statistics() → None

Resets all execution and total time counters. Affects the results of get_total_time(), get_total_execution_time() and get_total_time_startpoint()

property resource_name: str

Returns the resource name used in the constructor.

self_test(*timeout: int = None*) → Tuple[int, str]

SCPI command: *TST? Performs instrument's self-test. Returns tuple (code:int, message: str). Code 0 means the self-test passed. You can define the custom timeout in milliseconds. If you do not define it, the method uses default self-test timeout (usually 60 secs).

send_file_from_pc_to_instrument(*source_pc_file: str*, *target_instr_file: str*) → None

SCPI Command: MMEM:DATA

Sends file from PC to the instrument.

classmethod set_global_logging_relative_time_of_first_entry() → None

This method sets the global flag, that takes the very first log entry start time of any global instance as a global relative timestamp. This means, after this call, the first instance that logs an entry set the relative timestamp for all global target instances, and begins with the start timestamp 0:00:00.000

classmethod set_global_logging_relative_timestamp(*timestamp: datetime*) → None

Sets global common relative timestamp for log entries. To use it, call the following: io.logger.set_relative_timestamp_global()

classmethod set_global_logging_relative_timestamp_now() → None

Sets global common relative timestamp for log entries to this moment. To use it, call the following: io.logger.set_relative_timestamp_global().

classmethod set_global_logging_target(*target*) → None

Sets global common target stream that each instance can use. To use it, call the following: io.logger.set_logging_target_global(). If an instance uses global logging target, it automatically uses the global relative timestamp (if set). You can set the target to None to invalidate it.

property supported_models: List[str]

Returns a list of the instrument models supported by this instrument driver

unlock_resource() → None

Unlocks the instrument to other clients.

property visa_manufacturer: str

Returns the manufacturer of the current VISA session.

property visa_timeout: int

Sets / returns visa IO timeout in milliseconds.

visa_tout_suppressor(*visa_tout_ms: int = 0*) → VisaTimeoutSuppressor

Returns Context Manager that suppresses the VISA timeout error. Careful!!!: Only the very first VISA Timeout exception is suppressed, and afterward the context ends. Therefore, use only one command per context manager, if you do not want to skip the following ones. :param visa_tout_ms: VISA Timeout in milliseconds, that is set for this context. Afterward, it is changed back. Default value: do-not-change.

write(*cmd: str*) → None

Writes the command to the instrument as string. This method is an alias to the write_str() method.

write_bin_block(*cmd: str*, *payload: bytes*) → None

Writes all the payload as binary data block to the instrument. The binary data header is added at the beginning of the transmission automatically, do not include it in the payload!!!

write_bin_block_from_file(*cmd: str*, *file_path: str*) → None

Writes data from the file as binary data block to the instrument using the provided command. Example for transferring a file from PC -> Instrument: cmd = f"MMEM:DATA '{INSTR_FILE_PATH}',".

Alternatively, use the dedicated methods for this purpose:

- `send_file_from_pc_to_instrument()`
- `read_file_from_instrument_to_pc()`

write_bool(*cmd: str*, *param: bool*) → None

Writes the command to the instrument followed by the boolean parameter: e.g.: cmd = 'OUTPUT' param = 'True', result command = 'OUTPUT ON'

write_bool_with_opc(*cmd: str*, *param: bool*, *timeout: int = None*) → None

Writes the command with OPC to the instrument followed by the boolean parameter: e.g.: cmd = 'OUTPUT' param = 'True', result command = 'OUTPUT ON' If you do not provide timeout, the method uses current opc_timeout.

write_float(*cmd: str*, *param: float*) → None

Writes the command to the instrument followed by the boolean parameter: e.g.: cmd = 'CENTER:FREQ' param = '10E6', result command = 'CENTER:FREQ 10E6'

write_float_with_opc(*cmd: str*, *param: float*, *timeout: int = None*) → None

Writes the command with OPC to the instrument followed by the boolean parameter: e.g.: cmd = 'CENTER:FREQ' param = '10E6', result command = 'CENTER:FREQ 10E6' If you do not provide timeout, the method uses current opc_timeout.

write_int(*cmd: str*, *param: int*) → None

Writes the command to the instrument followed by the integer parameter: e.g.: cmd = 'SELECT:INPUT' param = '2', result command = 'SELECT:INPUT 2'

write_int_with_opc(*cmd: str*, *param: int*, *timeout: int = None*) → None

Writes the command with OPC to the instrument followed by the integer parameter: e.g.: cmd = 'SELECT:INPUT' param = '2', result command = 'SELECT:INPUT 2' If you do not provide timeout, the method uses current opc_timeout.

write_str(*cmd: str*) → None

Writes the command to the instrument as string. This method is an alias to write() method.

write_str_with_opc(*cmd: str*, *timeout: int = None*) → None

Writes the opc-synced command to the instrument. If you do not provide timeout, the method uses current opc_timeout.

write_with_opc(*cmd: str*, *timeout: int = None*) → None

This method is an alias to the write_str_with_opc(). Writes the opc-synced command to the instrument. If you do not provide timeout, the method uses current opc_timeout.

20.3 Module contents

VISA communication interface for SCPI-based instrument remote control. :version: 1.122.0 :copyright: 2025 by Rohde & Schwarz GMBH & Co. KG :license: MIT, see LICENSE for more details.

class BinFloatFormat(*value*)

Bases: Enum

Binary format of a float number.

Double_8bytes = 3

Double_8bytes_swapped = 4

Single_4bytes = 1

Single_4bytes_swapped = 2

class BinIntFormat(*value*)

Bases: Enum

Binary format of an integer number.

Integer16_2bytes = 3

Integer16_2bytes_swapped = 4

Integer32_4bytes = 1

Integer32_4bytes_swapped = 2

exception DriverValueError(*rsrc_name: str*, *message: str*)

Bases: *RsInstrException*

Exception for different driver value settings e.g. RepCap values or Enum values.

class IoTransferEventArgs(*reading: bool*, *opc_sync: bool*, *total_size: int | None*, *context: str*)

Bases: object

Contains event data for driver read or write operations.

Initializes new instance of IoTransferEventArgs :param reading: True: reading operation, False: writing operation :param opc_sync: defines if the command is OPC-synchronised :param total_size: total size of the data received :param context: SCPI query. It is truncated to maximum of 100 characters

binary: bool

string data

Type
True

Type
Binary data, False

chunk_ix: int

0-based index of the chunk.

chunk_size: int

Size of one chunk of data. This number does not change during the transfer.

data: str | bytes

If the feature of transferring data over R/W event is switched ON, this field contains the whole data.

id_generator = count(100)

classmethod read_chunk(*opc_sync: bool*, *context: str*) → *IoTransferEventArgs*

Creates new IoTransferEventArgs of read string

Parameters

- **opc_sync** – defines if the command is OPC-synchronised
- **context** – SCPI query. It is truncated to maximum of 100 characters.

Returns

IoTransferEventArgs object of a read string operation.

resource_name: str

Visa Resource Name of the instrument that generated the data.

set_end_of_transfer()

Sets fields to signal end of transfer.

total_chunks: int

Expected number of chunks.

property transfer_id: int

Unique number for each transfer for this Instrument. If the transfer is performed in more chunks, the transfer_id stays the same during the whole transfer.

classmethod write_bin(*context: str*) → *IoTransferEventArgs*

Creates new IoTransferEventArgs of read binary data

Parameters

context – SCPI command. It is truncated to maximum of 100 characters.

Returns

IoTransferEventArgs object of a write-binary-data operation.

classmethod write_str(*opc_sync: bool*, *total_size: int*, *context: str*) → *IoTransferEventArgs*

Creates new IoTransferEventArgs of write string

Parameters

- **opc_sync** – defines if the command is OPC-synchronised
- **total_size** – size of the data to write
- **context** – SCPI command write. It is truncated to maximum of 100 characters

Returns

IoTransferEventArgs object of a write-string operation.

class LoggingMode(*value*)

Bases: Enum

Determines the format of the logging message.

Default = 3

Errors = 2

Off = 0

On = 1

class OpcSyncQueryMechanism(*value*)

Bases: Enum

Mechanism to use when querying with OPC.

also_check_mav = 1

cls_only_check_mav_err_queue = 2

only_check_mav_err_queue = 3

standard = 0

exception ResourceError(*rsrc_name: str*, *message: str*)

Bases: *RsInstrException*

Exception for resource name - e.g. resource not found.

exception RsInstrException(*message: str*)

Bases: Exception

Exception base class for all the RsInstrument exceptions.

class RsInstrument(*resource_name: str*, *id_query: bool = True*, *reset: bool = False*, *options: str = None*, *direct_session: object = None*)

Bases: object

Root class for remote-controlling instrument with SCPI commands.

Initializes new RsInstrument session.

Parameters

- **resource_name** – VISA resource name, e.g. 'TCPIP::192.168.2.1::INSTR'
- **id_query** – if True, the instrument's model name is verified against the models supported by the driver and eventually throws an exception
- **reset** – Resets the instrument (sends *RST) command and clears its status syb-system
- **direct_session** – Another driver object or pyVisa object to reuse the session instead of opening a new session
- **options** – string tokens alternating the driver settings. More tokens are separated by comma.

Parameter options tokens examples:

- Simulate=True - starts the session in simulation mode. Default: False
- SelectVisa=socketio - uses no VISA implementation for socket connections - you do not need any VISA-C installation
- SelectVisa=rs - forces usage of RohdeSchwarz Visa
- SelectVisa=ni - forces usage of National Instruments Visa
- Profile = HM8123 - setting profile fitting the specific non-standard instruments. Available values: HM8123, CMQ, ATS, Minimal. Default: none
- OpenTimeout=5000 - sets timeout used at the session opening. This timeout is only used in waiting for a locked session to be freed. Default: 2000ms

- `ExclusiveLock=True` - opens the session with exclusive lock on the VISA level. Default: `False`
- `QueryInstrumentStatus = False` - same as `driver.utilities.instrument_status_checking = False`. Default: `True`
- `WriteDelay = 20, ReadDelay = 5` - introduces delay of 20ms before each write and 5ms before each read. Default: `0ms` for both
- `TerminationCharacter = "\r"` - sets the termination character for reading. Default: `\n` (LineFeed or LF)
- `AssureWriteWithTermChar = True` - makes sure each command/query is terminated with termination character. Default: Interface dependent
- `AddTermCharToWriteBinBlock = True` - adds one additional LF to the end of the binary data (some instruments require that). Default: `False`
- `DataChunkSize = 10E3` - maximum size of one write/read segment. If transferred data is bigger, it is split to more segments. Default: `1E7` bytes
- `OpcTimeout = 10000` - same as driver.utilities.opc_timeout = 10000. Default: `30000ms`
- `VisaTimeout = 5000` - same as driver.utilities.visa_timeout = 5000. Default: `10000ms`
- `ViClearExeMode = Disabled` - viClear() execution mode. Default: `execute_on_all`
- `OpcQueryAfterWrite = True` - same as driver.utilities.opc_query_after_write = True. Default: `False`
- `OpcWaitMode = OpcQuery` - mode for all the opc-synchronised write/reads. Other modes: StbPolling, StbPollingSlow, StbPollingSuperSlow. Default: `StbPolling`
- `StbInErrorCheck = False` - if true, the driver checks errors with *STB? If false, it uses SYST:ERR?. Default: `True`
- `SkipStatusSystemSettings = False` - some instruments do not support full status system commands. In such case, set this value to True. Default: `False`
- `SkipClearStatus = True` - set to True for instruments that do not support *CLS command. Default: `False`
- `DisableOpcQuery = True` - set to True for instruments that do not support *OPC? query. Default: `False`
- `EachCmdAsQuery = True`, set to True, for instruments that always return answer. Default: `false`
- `CmdIdn = ID?` - defines which SCPI command to use for identification query. Use '<none>' string to skip identification query at the init. Default: `*IDN?`
- `CmdReset = RT` - defines which SCPI command to use for reset. Default: `*RST`
- `VxiCapable = false` - you can force a session to a VXI-incapable. Default: <interface-dependent>
- `Encoding = utf-8` - setting of encoding for strings into bytes and vice versa. Default: `charmap`
- `OpcSyncQueryMechanism = AlsoCheckMav` - setting of mechanism for OPC-synchronised queries. Default: `OnlyCheckMavErrQueue`
- `FirstCmds = *CLS` - first command(s) to sent after init. Separated more commands/queries with ';;'. Default: ````
- `EachCmdPrefix = lf` - this prefix is added to the beginning of each command sent to the instrument. Default: ````
- `EachCmdSuffix = cr` - this suffix is added to the end of each command sent to the instrument. Default: ````

- `StripStringTrailingWhitespaces = True` - use it to strip white spaces from string query responses. Default: `False`
- `LoggingMode = On` - sets the logging status right from the start. Possible values: On | Off | Error. Default: `Off`
- `LoggingName = 'MyDevice'` - sets the name to represent the session in the log entries. Default: `<resource_name>`
- `LoggingFormat = 'PAD_LEFT12(%START_TIME%) PAD_LEFT25(%DEVICE_NAME%) PAD_LEFT12(%DURATION%) %SCPI_COMMAND%'` - sets the format of the log entries. Default: `PAD_LEFT12(%START_TIME%) PAD_LEFT25(%DEVICE_NAME%) PAD_LEFT12(%DURATION%) %LOG_STRING_INFO%: %LOG_STRING%`
- `LogToGlobalTarget = True` - sets the logging target to the class-property previously set with RsInstrument.set_global_logging_target() Default: `False`
- `LoggingToConsole = True` - immediately starts logging to the console. Default: False
- `LoggingToUdp = True` - immediately starts logging to the UDP port. Default: False
- `LoggingUdpPort = 49200` - UDP port to log to. Default: 49200
- `LoggingRelativeTimeOfFirstEntry = True` - Logging starts with relative time set to the first log entry, which causes the first start time to be '00:00:00.000'. Default: False

add_instr_option(*option: str*) → None

Adds new option if not already existing.

static assert_minimum_version(*min_version: str*) → None

Asserts that the driver version fulfills the minimum required version you have entered. This way you make sure your installed driver is of the entered version or newer.

assign_lock(*lock: RLock*) → None

Assigns the provided thread lock.

property bin_float_numbers_format: *BinFloatFormat*

Sets / returns format of float numbers when transferred as binary data

property bin_int_numbers_format: *BinIntFormat*

Sets / returns format of integer numbers when transferred as binary data

check_status() → None

Throws InstrumentStatusException in case of an error in the instrument's error queue. The status checking is performed always, independent of the property 'instrument_status_checking'. Also, the property ScpiLogger.log_status_check_ok is ignored, and the Status check is always logged.

classmethod clear_global_logging_relative_timestamp() → None

Clears the global relative timestamp. After this, all the instances using the global relative timestamp continue logging with the absolute timestamps.

clear_lock() → None

Clears the existing thread lock, making the current session thread-independent from others that might share the current thread lock.

clear_status() → None

Clears instrument's status system, the session's I/O buffers and the instrument's error queue

close() → None

Closes the active RsInstrument session

property data_chunk_size: int

Returns max chunk size of one data block.

property driver_version: str

Returns the RsInstrument package version

property encoding: str

Returns string<=>bytes encoding of the session.

property events: *Events*

Interface for event handlers, see *here*

file_exists(*instr_file: str*) → bool

Returns true, if the instrument file exist.

classmethod from_existing_session(*session: object*, *options: str = None*) → *RsInstrument*

Creates a new RsInstrument object with the entered 'session' reused. :param session: can be another driver or a direct pyvisa session. :param options: string tokens alternating the driver settings. More tokens are separated by comma.

property full_instrument_model_name: str

Returns the current instrument's full name e.g. 'FSW26'

classmethod get_driver_version() → str

Returns the RsInstrument package version

get_file_size(*instr_file: str*) → int | None

Return size of the instrument file, or None if the file does not exist.

classmethod get_global_logging_relative_timestamp() → datetime | None

Returns global common relative timestamp for log entries.

classmethod get_global_logging_target()

Returns global common target stream.

get_last_sent_cmd() → str

Returns the last commands sent to the instrument. Only works in simulation mode.

get_lock() → RLock

Returns the thread lock for the current session.

By default:

- If you create a new RsInstrument instance with new VISA session, the session gets a new thread lock. You can assign it to another RsInstrument sessions in order to share one physical instrument with a multi-thread access.
- If you create a new RsInstrument from an existing session, the thread lock is shared automatically making both instances multi-thread safe.

You can always assign new thread lock by calling `driver.utilities.assign_lock()`

get_session_handle()

Returns the underlying pyvisa session

get_total_execution_time() → timedelta

Returns total time spent by the library on communicating with the instrument. This time is always shorter than get_total_time(), since it does not include gaps between the communication. You can reset this counter with reset_time_statistics().

get_total_time() → timedelta

Returns delta time spent by the library between the get_total_time_startpoint() and now. This time is always longer than get_total_execution_time(), since it also includes all other activities besides the communication. You can set the total time startpoint to now with reset_time_statistics().

get_total_time_startpoint() → datetime

Returns time from which the execution started. This is the value that the get_total_time() calculates as its reference. Calling the reset_time_statistics() sets this time to now.

go_to_local(*mixed_mode: bool = True*) → None

Puts the instrument into local state. By default, the method uses a mechanism to keep the instrument in a mixed mode: remote and local. That means, you can remote-control your instrument, and at the same time it still allows manual control. Set the mixed_mode to False, if you want your instrument to go to remote mode as soon as it receives the first remote command.

go_to_remote() → None

Puts the instrument into remote state.

has_instr_option(*options: str | List[str]*) → bool

Returns true, if the entered options (case-insensitive) matches at least one of the installed options (or-logic). You can enter either a string with one option, or more options '/'-separated, or more options as a list of strings. If K0 is present, all the K-options are reported as present. B-options are not affected by K0. Example 1: options='k23' returns true, if the instrument has the option 'K23'. Example 2: options='k23 / K23e' returns true, if the instrument has either the option 'K23' or the option 'K23E'. Example 3: options=['k11','K22'] returns true, if the instrument has either the option 'K11' or the option 'K22'.

has_instr_option_k0() → bool

Returns true, if the instrument has K0 installed.

has_instr_option_regex(*re_options: str | List[str]*) → bool

Returns true, if the entered regex string (case-insensitive) matches at least one of the installed options. The match must be complete, not just partial (search). You can enter either a string with one option, or more options '/'-separated, or more options as a list of strings. Example 1: re_options='k10.' returns true, if the instrument contains any option 'K100' … up to 'K109' . Example 2: re_options='k10. / k20.*' returns true, if the instrument contains any of the options 'K10x' or 'K20xxx'. Example 3: re_options=['k10.', 'k20.*'] returns true, if the instrument contains any options 'K10x' or 'K20xxx'.

property idn_string: str

Returns instrument's identification string - the response on the SCPI command *IDN?

instr_err_suppressor(*visa_tout_ms: int = 0*, *suppress_only_codes: int | List[int] = None*) → InstrErrorSuppressor

Returns Context Manager that suppresses the instrument errors. Other exceptions types are still raised. On entering the context, this class clears all the instrument status errors. :param visa_tout_ms: VISA Timeout in milliseconds, that is set for this context. Afterward, it is changed back. Default value: do-not-change. :param suppress_only_codes: You can enter a code or list of codes for errors to be suppressed. Other errors will be reported. Example: If you enter -113 here, only the 'Undefined Header' error will be suppressed. Default value: suppress-all-errors.

property instrument_firmware_version: str

Returns instrument's firmware version

property instrument_model_name: str

Returns the current instrument's family name e.g. 'FSW'

property instrument_options: List[str]

Returns all the instrument options. The options are sorted in the ascending order starting with K-options and continuing with B-options

property instrument_serial_number: str

Returns instrument's serial_number

property instrument_status_checking: bool

Sets / returns Instrument Status Checking. When True (default is True), all the driver methods and properties are sending "SYSTem:ERRor?" at the end to immediately react on error that might have occurred. We recommend keeping the state checking ON all the time. Switch it OFF only in rare cases when you require maximum speed. The default state after initializing the session is ON.

is_connection_active() → bool

Returns true, if the VISA connection is active and the communication with the instrument still works. WARNING!!! this method queries the session's VISA Timeout and additionally, queries the *IDN? from the instrument, hence affects the performance of your application when used regularly.

static list_resources(*expression: str = '?*::INSTR'*, *visa_select: str = None*) → List[str]

Finds all the resources defined by the expression.

- '?*' - matches all the available instruments
- 'USB::?*' - matches all the USB instruments
- 'TCPIP::192?*' - matches all the LAN instruments with the IP address starting with 192

Parameters

- **expression** – see the examples in the function
- **visa_select** – optional parameter selecting a specific VISA. Examples: '@ivi', '@rs'

lock_resource(*timeout: int*, *requested_key: str | bytes = None*) → bytes | str

Locks the instrument to prevent it from communicating with other clients.

property logger: *ScpiLogger*

Scpi Logger interface, see *here*

property manufacturer: str

Returns manufacturer of the instrument

property opc_query_after_write: bool

Sets / returns Instrument *OPC? query sending after each command write. When True, (default is False) the driver sends *OPC? every time a write command is performed. Use this if you want to make sure your sequence is performed command-after-command.

property opc_sync_query_mechanism: *OpcSyncQueryMechanism*

Returns the current setting of the OPC-Sync query mechanism.

property opc_timeout: int

Sets / returns timeout in milliseconds for all the operations that use OPC synchronization.

process_all_commands() → None

SCPI command: *WAI Stops further commands processing until all commands sent before *WAI have been executed.

query(*query: str*) → str

Sends the string query to the instrument and returns the response as string. The response is trimmed of any trailing LF characters and has no length limit. This method is an alias to the query_str() method.

query_all_errors() → List[str] | None

Queries and clears all the errors from the instrument's error queue. The method returns list of strings as error messages. If no error is detected, the return value is None. The process is: querying 'SYSTem:ERRor?' in a loop until the error queue is empty. If you want to include the error codes, call the query_all_errors_with_codes()

query_all_errors_with_codes() → List[Tuple[int, str]] | None

Queries and clears all the errors from the instrument's error queue. The method returns list of tuples (code: int, message: str). If no error is detected, the return value is None. The process is: querying 'SYSTem:ERRor?' in a loop until the error queue is empty.

query_bin_block(*query: str*) → bytes

Queries binary data block to bytes. Throws an exception if the returned data was not a binary data. Returns data:bytes

query_bin_block_to_file(*query: str*, *file_path: str*, *append: bool = False*) → None

Queries binary data block to the provided file. If append is False, any existing file content is discarded. If append is True, the new content is added to the end of the existing file, or if the file does not exit, it is created. Throws an exception if the returned data was not a binary data. Example for transferring a file from Instrument -> PC: query = f"MMEM:DATA? '{INSTR_FILE_PATH}'".

Alternatively, use the dedicated methods for this purpose:

- `send_file_from_pc_to_instrument()`
- `read_file_from_instrument_to_pc()`

query_bin_block_to_file_with_opc(*query: str*, *file_path: str*, *append: bool = False*, *timeout: int = None*) → None

Sends a OPC-synced query and writes the returned data to the provided file. If append is False, any existing file content is discarded. If append is True, the new content is added to the end of the existing file, or if the file does not exit, it is created. Throws an exception if the returned data was not a binary data.

query_bin_block_with_opc(*query: str*, *timeout: int = None*) → bytes

Sends a OPC-synced query and returns binary data block to bytes. If you do not provide timeout, the method uses current opc_timeout.

query_bin_or_ascii_float_list(*query: str*) → List[float]

Queries a list of floating-point numbers that can be returned as ASCII or binary format.

- For ASCII format, the list numbers are decoded as comma-separated values.
- For Binary Format, the numbers are decoded based on the property BinFloatFormat, usually float 32-bit (FORM REAL,32).

query_bin_or_ascii_float_list_with_opc(*query: str*, *timeout: int = None*) → List[float]

Sends a OPC-synced query and reads a list of floating-point numbers that can be returned as ASCII or binary format.

- For ASCII format, the list numbers are decoded as comma-separated values.
- For Binary Format, the numbers are decoded based on the property BinFloatFormat, usually float 32-bit (FORM REAL,32).

If you do not provide timeout, the method uses current opc_timeout.

query_bin_or_ascii_int_list(*query: str*) → List[int]

Queries a list of floating-point numbers that can be returned as ASCII or binary format.

- For ASCII format, the list numbers are decoded as comma-separated values.
- For Binary Format, the numbers are decoded based on the property BinFloatFormat, usually float 32-bit (FORM REAL,32).

query_bin_or_ascii_int_list_with_opc(*query: str*, *timeout: int = None*) → List[int]

Sends a OPC-synced query and reads a list of floating-point numbers that can be returned as ASCII or binary format.

- For ASCII format, the list numbers are decoded as comma-separated values.
- For Binary Format, the numbers are decoded based on the property BinFloatFormat, usually float 32-bit (FORM REAL,32).

If you do not provide timeout, the method uses current opc_timeout.

query_bool(*query: str*) → bool

Sends the query to the instrument and returns the response as boolean.

query_bool_list(*query: str*) → List[bool]

Sends the string query to the instrument and returns the response as List of booleans, where the delimiter is comma (‘,’). Blank or empty response is returned as an empty list.

query_bool_list_with_opc(*query: str*, *timeout: int = None*) → List[bool]

Sends a OPC-synced query and reads response from the instrument as csv-list of booleans. If you do not provide timeout, the method uses current opc_timeout. Blank or empty response is returned as an empty list.

query_bool_with_opc(*query: str*, *timeout: int = None*) → bool

Sends the opc-synced query to the instrument and returns the response as boolean. If you do not provide timeout, the method uses current opc_timeout.

query_float(*query: str*) → float

Sends the query to the instrument and returns the response as float.

query_float_with_opc(*query: str*, *timeout: int = None*) → float

Sends the opc-synced query to the instrument and returns the response as float. If you do not provide timeout, the method uses current opc_timeout.

query_int(*query: str*) → int

Sends the query to the instrument and returns the response as integer.

query_int_with_opc(*query: str*, *timeout: int = None*) → int

Sends the opc-synced query to the instrument and returns the response as integer. If you do not provide timeout, the method uses current opc_timeout.

query_opc(*timeout: int = 0*) → int

SCPI command: *OPC? Queries the instrument’s OPC bit and hence it waits until the instrument reports operation complete. If you define timeout > 0, the VISA timeout is set to that value just for this method call.

query_str(*query: str*) → str

Sends the string query to the instrument and returns the response as string. The response is trimmed of any trailing LF characters and has no length limit. This method is an alias to the query() method.

query_str_list(*query: str*, *remove_blank_response: bool = False*) → List[str]

Sends the string query to the instrument and returns the response as List of strings, where the delimiter is comma (‘,’). Each element of the list is trimmed for leading and trailing quotes.

Meaning of the ‘remove_blank_response’:

- False(default): whitespaces-only response is returned as a list with one empty element [‘’].
- True: whitespaces-only response is returned as an empty list [].

query_str_list_with_opc(*query: str*, *timeout: int = None*, *remove_blank_response: bool = False*) → List[str]

Sends a OPC-synced query and reads response from the instrument as csv-list.
If you do not provide timeout, the method uses current opc_timeout.

Meaning of the ‘remove_blank_response’:

- False(default): whitespaces-only response is returned as a list with one empty element [‘’].
- True: whitespaces-only response is returned as an empty list [].

query_str_stripped(*query: str*) → str

Sends the string query to the instrument and returns the response as string stripped of the trailing LF and leading/trailing single/double quotes. The stripping of the leading/trailing quotes is blocked, if the string contains the quotes in the middle. This method is an alias to the query_stripped() method.

query_str_with_opc(*query: str*, *timeout: int = None*) → str

Sends the opc-synced query to the instrument and returns the response as string. The response is trimmed of any trailing LF characters and has no length limit. If you do not provide timeout, the method uses current opc_timeout.

query_stripped(*query: str*) → str

Sends the string query to the instrument and returns the response as string stripped of the trailing LF and leading/trailing single/double quotes. The stripping of the leading/trailing quotes is blocked, if the string contains the quotes in the middle.

query_with_opc(*query: str*, *timeout: int = None*) → str

This method is an alias to the write_str_with_opc(). Sends the opc-synced query to the instrument and returns the response as string. The response is trimmed of any trailing LF characters and has no length limit. If you do not provide timeout, the method uses current opc_timeout.

read_file_from_instrument_to_pc(*source_instr_file: str*, *target_pc_file: str*, *append_to_pc_file: bool = False*) → None

SCPI Command: MMEM:DATA?

Reads file from instrument to the PC.

Set the `append_to_pc_file` to True if you want to append the read content to the end of the existing PC file.

reconnect(*force_close: bool = False*) → bool

If the connection is not active, the method tries to reconnect to the device. If the connection is active, and force_close is False, the method does nothing. If the connection is active, and force_close is True, the method closes, and opens the session again. Returns True, if the reconnection has been performed.

remove_instr_option(*option: str*) → None

Removes the option if exists.

reset(*timeout: int = 0*) → None

SCPI command: *RST Sends *RST command + calls the clear_status(). If you define timeout > 0, the VISA timeout is set to that value just for this method call.

reset_time_statistics() → None

Resets all execution and total time counters. Affects the results of get_total_time(), get_total_execution_time() and get_total_time_startpoint()

property resource_name: str

Returns the resource name used in the constructor.

self_test(*timeout: int = None*) → Tuple[int, str]

SCPI command: *TST? Performs instrument's self-test. Returns tuple (code:int, message: str). Code 0 means the self-test passed. You can define the custom timeout in milliseconds. If you do not define it, the method uses default self-test timeout (usually 60 secs).

send_file_from_pc_to_instrument(*source_pc_file: str*, *target_instr_file: str*) → None

SCPI Command: MMEM:DATA

Sends file from PC to the instrument.

classmethod set_global_logging_relative_time_of_first_entry() → None

This method sets the global flag, that takes the very first log entry start time of any global instance as a global relative timestamp. This means, after this call, the first instance that logs an entry set the relative timestamp for all global target instances, and begins with the start timestamp 0:00:00.000

classmethod set_global_logging_relative_timestamp(*timestamp: datetime*) → None

Sets global common relative timestamp for log entries. To use it, call the following: io.logger.set_relative_timestamp_global()

classmethod set_global_logging_relative_timestamp_now() → None

Sets global common relative timestamp for log entries to this moment. To use it, call the following: io.logger.set_relative_timestamp_global().

classmethod set_global_logging_target(*target*) → None

Sets global common target stream that each instance can use. To use it, call the following: io.logger.set_logging_target_global(). If an instance uses global logging target, it automatically uses the global relative timestamp (if set). You can set the target to None to invalidate it.

property supported_models: List[str]

Returns a list of the instrument models supported by this instrument driver

unlock_resource() → None

Unlocks the instrument to other clients.

property visa_manufacturer: str

Returns the manufacturer of the current VISA session.

property visa_timeout: int

Sets / returns visa IO timeout in milliseconds.

visa_tout_suppressor(*visa_tout_ms: int = 0*) → VisaTimeoutSuppressor

Returns Context Manager that suppresses the VISA timeout error. Careful!!!: Only the very first VISA Timeout exception is suppressed, and afterward the context ends. Therefore, use only one command per context manager, if you do not want to skip the following ones. :param visa_tout_ms: VISA Timeout in milliseconds, that is set for this context. Afterward, it is changed back. Default value: do-not-change.

write(*cmd: str*) → None

Writes the command to the instrument as string. This method is an alias to the write_str() method.

write_bin_block(*cmd: str*, *payload: bytes*) → None

Writes all the payload as binary data block to the instrument. The binary data header is added at the beginning of the transmission automatically, do not include it in the payload!!!

write_bin_block_from_file(*cmd: str*, *file_path: str*) → None

Writes data from the file as binary data block to the instrument using the provided command. Example for transferring a file from PC -> Instrument: cmd = f"MMEM:DATA '{INSTR_FILE_PATH}',".

Alternatively, use the dedicated methods for this purpose:

- `send_file_from_pc_to_instrument()`
- `read_file_from_instrument_to_pc()`

write_bool(*cmd: str*, *param: bool*) → None

Writes the command to the instrument followed by the boolean parameter: e.g.: cmd = 'OUTPUT' param = 'True', result command = 'OUTPUT ON'

write_bool_with_opc(*cmd: str*, *param: bool*, *timeout: int = None*) → None

Writes the command with OPC to the instrument followed by the boolean parameter: e.g.: cmd = 'OUTPUT' param = 'True', result command = 'OUTPUT ON' If you do not provide timeout, the method uses current opc_timeout.

write_float(*cmd: str*, *param: float*) → None

Writes the command to the instrument followed by the boolean parameter: e.g.: cmd = 'CENTER:FREQ' param = '10E6', result command = 'CENTER:FREQ 10E6'

write_float_with_opc(*cmd: str*, *param: float*, *timeout: int = None*) → None

Writes the command with OPC to the instrument followed by the boolean parameter: e.g.: cmd = 'CENTER:FREQ' param = '10E6', result command = 'CENTER:FREQ 10E6' If you do not provide timeout, the method uses current opc_timeout.

write_int(*cmd: str*, *param: int*) → None

Writes the command to the instrument followed by the integer parameter: e.g.: cmd = 'SELECT:INPUT' param = '2', result command = 'SELECT:INPUT 2'

write_int_with_opc(*cmd: str*, *param: int*, *timeout: int = None*) → None

Writes the command with OPC to the instrument followed by the integer parameter: e.g.: cmd = 'SELECT:INPUT' param = '2', result command = 'SELECT:INPUT 2' If you do not provide timeout, the method uses current opc_timeout.

write_str(*cmd: str*) → None

Writes the command to the instrument as string. This method is an alias to write() method.

write_str_with_opc(*cmd: str*, *timeout: int = None*) → None

Writes the opc-synced command to the instrument. If you do not provide timeout, the method uses current opc_timeout.

write_with_opc(*cmd: str*, *timeout: int = None*) → None

This method is an alias to the write_str_with_opc(). Writes the opc-synced command to the instrument. If you do not provide timeout, the method uses current opc_timeout.

exception StatusException(*rsrc_name: str*, *message: str*, *errors_list: List[Tuple[int, str]]*, *first_exc: type = None*)

Bases: *RsInstrException*

Exception for instrument status errors. Tje field errors_list contains the complete list of all the errors with messages and codes.

exception TimeoutException(*message: str*)

Bases: *RsInstrException*

Exception for timeout errors.

exception UnexpectedResponseException(*rsrc_name: str*, *message: str*)

Bases: *RsInstrException*

Exception for instrument unexpected responses.

size_to_kb_mb_gb_string(*data_size: int*, *as_additional_info: bool = False*, *allow_gb: bool = True*) → str

Returns human-readable string with kilobytes, megabytes or gigabytes depending on the data_size range.

param data_size
data size in bytes to convert

param allow_gb
allow also Gigabytes size

param as_additional_info

if True, the dynamic data appear in round bracket after the number in bytes. e.g. '12345678 bytes (11.7 MB)' if False, only the dynamic data is returned e.g. '11.7 MB'

size_to_kb_mb_string(*data_size: int*, *as_additional_info: bool = False*) → str

Returns human-readable string with kilobytes or megabytes depending on the data_size range.

Parameters

- **data_size** – data size in bytes to convert
- **as_additional_info**

if True, the dynamic data appear in round bracket after the number in bytes. e.g. '12345678 bytes (11.7 MB)' if False, only the dynamic data is returned e.g. '11.7 MB'

value_to_si_string(*value: float*, *fmt: str = '.12g'*, *min_decimal_places: int = 0*, *str_after_number: str = ' '*) → str

Returns the entered float value converted to string with SI notation. :param value: input float value :param fmt: formatting of the number. Default: '.12g' :param min_decimal_places: assures minimal number of decimal places. If you set the number > 0, the function makes sure the decimal places are kept, and if there are less of them, they are filled with nulls ('.000' or '000') Default: 0 .. rubric:: Examples

- value = 1, fmt = '.12g', min_decimal_places = 0, result -> '1'
- value = 1, fmt = '.12g', min_decimal_places = 3, result -> '1.000'

Parameters
str_after_number – string after number, if the SI suffix is present. Default: ' '

Examples

- 1.23 -> '1.23'
- 1.23E3 -> '1.23 k'
- 2.56E9 -> '2.56 G'

- 11.3E-6 -> '-11.3 u'

This function is useful for user-readable string outputs.

CHAPTER

TWENTYONE

RSINSTRUMENT.LOGGER

Check the usage in the Getting Started chapter *Logging*.

class ScpiLogger

Base class for SCPI logging

mode

Sets the logging ON or OFF. Additionally, you can set the logging ON only for errors. Possible values:

- LoggingMode.Off - logging is switched OFF
- LoggingMode.On - logging is switched ON
- LoggingMode.Errors - logging is switched ON, but only for error entries
- LoggingMode.Default - sets the logging to default - the value you have set with logger.default_mode

stop() → None

Stops the logging. This is the same as: mode = LoggingMode.Off

start() → None

Starts the logging with the last defined LoggingMode. Default is LoggingMode.On

default_mode

Sets / returns the default logging mode. You can recall the default mode by calling the 'logger.mode = LoggingMode.Default'.

device_name: str

Use this property to change the resource name in the log from the default Resource Name (e.g. TCPIP::192.168.2.101::INSTR) to another name e.g. 'MySigGen1'.

set_logging_target(*target*, *console_log: bool | None = None*, *udp_log: bool | None = None*) → None

Sets local logging stream target - the target must implement write() and flush(). You can optionally set the console and UDP logging ON or OFF. This method switches the logging target global to OFF.

get_logging_target()

Based on the global_mode, it returns the logging target: either the local or the global one.

set_logging_target_global(*console_log: bool | None = None*, *udp_log: bool | None = None*) → None

Sets logging target to global. The global target must be defined. You can optionally set the console and UDP logging ON or OFF. This method switches the logging target global to ON.

log_to_console

Returns logging to console status.

log_to_udp

Returns logging to UDP status.

log_to_console_and_udp

Returns true, if both logging to UDP and console in are True.

info_raw(*log_entry: str*, *add_new_line: bool = True*) → None

Method for logging the raw string without any formatting.

info(*start_time: datetime | float | None*, *end_time: datetime | float | None*, *log_string_info: str*, *log_string: str*, *cmd: str | None = None*) → None

Method for logging one info entry. For binary log_string, use the info_bin()

error(*start_time: datetime | float | None*, *end_time: datetime | float | None*, *log_string_info: str*, *log_string: str*, *cmd: str | None = None*) → None

Method for logging one error entry.

set_relative_timestamp(*timestamp: datetime*) → None

If set, the further timestamps will be relative to the entered time.

set_relative_timestamp_now() → None

Sets the relative timestamp to the current time.

get_relative_timestamp() → datetime | None

Based on the global_mode, it returns the relative timestamp: either the local or the global one.

clear_relative_timestamp() → None

Clears the reference time, and the further logging continues with absolute times.

set_time_offset_zero_on_first_entry() → None

Sets the reference time to the value of the first log entry start time. This effectively means, the log is guaranteed to start with "00:00:00.000" start time.

flush() → None

Flush all the entries.

sync_from(*source:* ScpiLogger) → None

Synchronises this Logger with the source logger.

log_status_check_ok

Sets / returns the current status of status checking OK. If True (default), the log contains logging of the status checking 'Status check: OK'. If False, the 'Status check: OK' is skipped - the log is more compact. Errors will still be logged.

clear_cached_entries() → None

Clears potential cached log entries. Cached log entries are generated when the Logging is ON, but no target has been defined yet.

set_format_string(*value: str*, *line_divider: str = '\n'*) → None

Sets new format string and line divider. If you just want to set the line divider, set the format string value=None The original format string is: `PAD_LEFT12(%START_TIME%) PAD_LEFT25(%DEVICE_NAME%) PAD_LEFT12(%DURATION%) %LOG_STRING_INFO%: %LOG_STRING%`

Additional variables to use: `%SCPI_COMMAND%`.

restore_format_string() → None

Restores the original format string and the line divider to LF

abbreviated_max_len_ascii: int

Defines the maximum length of one ASCII log entry. Default value is 200 characters.

abbreviated_max_len_bin: int

Defines the maximum length of one Binary log entry. Default value is 2048 bytes.

abbreviated_max_len_list: int

Defines the maximum length of one list entry. Default value is 100 elements.

bin_line_block_size: int

Defines number of bytes to display in one line. Default value is 16 bytes.

udp_port

Returns udp logging port.

target_auto_flushing

Returns status of the auto-flushing for the logging target.

log_info_replacer: *LogInfoReplacer*

Replacer for Log Info Strings.

class LogInfoReplacer

Replacer, that takes the SCPI Logger Log Info and customizes its value.

Create the new LogInfoReplacer either with empty replacer dictionaries, or the ones from another LogInfoReplacer.

set_full_replacer(*repl_dict: Dict[str, str]*) → None

Sets the full-replacer dictionary. This replacer searches the dictionary (case-sensitive) for the key that equals the LogInfoString, and replaces the whole info string with the string value associated with that key.

put_full_replacer_item(*match: str*, *replace: str*) → None

Sets/replaces one item in the full replacer.

pop_full_replacer_item(*match: str*) → None

Pops(removes) the entered full replacer item.

set_regex_sr_replacer(*repl_dict: Dict[Pattern, str]*) → None

Sets the regex search&replace replacer dictionary. The replacer uses the re.sub() method of each key and replaces the matched substring with the string value associated with that key.

put_regex_sr_replacer_item(*search: Pattern | str*, *replace: str*) → Pattern

Sets/replaces one item in the regex replacer. The key can be either a regex pattern, or a string. In case of the string, the key is compiled to a regex pattern, before it is put into the replacing dictionary. Returns the resulting search regex pattern.

pop_regex_sr_replacer_item(*search: Pattern | str*) → None

Pops(removes) the entered regex replacer item. The key can be either a regex pattern, or a string. In case of the string, the key is compiled to a regex pattern prior to popping.

clear_replacers()

Clears both full and regex replacer dictionaries.

CHAPTER

TWENTYTWO

RSINSTRUMENT.EVENTS

class Events

Common Events class. Event-related methods and properties. Here you can set all the event handlers.

property before_query_handler: Callable

Returns the handler of before_query events.

Returns
current `before_query_handler`

property before_write_handler: Callable

Returns the handler of before_write events.

Returns
current `before_write_handler`

property io_events_include_data: bool

Returns the current state of the io_events_include_data See the setter for more details.

property on_read_handler: Callable

Returns the handler of on_read events.

Returns
current `on_read_handler`

property on_write_handler: Callable

Returns the handler of on_write events.

Returns
current `on_write_handler`

sync_from(*source:* Events) → None

Synchronises these Events with the source.

CHAPTER

TWENTYTHREE

INDEX AND SEARCH

- genindex
- search

PYTHON MODULE INDEX

r

INDEX

H

I

L

M

O

P

Q

R

S

T

U

V

W