RsInstrument Python

Release 1.22.0.80

Rohde & Schwarz

Apr 21, 2022

CONTENTS:

CHAPTER

ONE

REVISION HISTORY

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

Check out the full documentation here: https://rsinstrument.readthedocs.io/

Examples: https://github.com/Rohde-Schwarz/Examples/tree/main/Misc/Python/RsInstrument https://github.com/Rohde-Schwarz/Examples/tree/main/Oscilloscopes/Python/RsInstrument https://github.com/Rohde-Schwarz/Examples/tree/main/Powersensors/Python/RsInstrument https://github.com/Rohde-Schwarz/Examples/tree/main/Powersupplies/Python/RsInstrument https://github.com/Rohde-Schwarz/Examples/tree/main/SpectrumAnalyzers/Python/RsInstrument https://github.com/Rohde-Schwarz/Examples/tree/main/VectorNetworkAnalyzers/Python/RsInstrument

Version history:

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: Driver-Setup=(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

1. 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 cool 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.

CHAPTER

FOUR

2. 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 bottom left
- Type `rsinstrument` in the search box
- Install the version 1.22.0 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

3. 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

4. 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 1.15.0 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('1.9.0')
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 2021 :-)

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::HISLIP', 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.

6.2 Selecting specific VISA

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

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

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-denier, 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::HISLIP', True, True, "Simulate=True")
```

More `option_string` tokens are separated by comma:

```
instr = RsInstrument('TCPIP::192.168.56.101::HISLIP', 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

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

Here, you may ask a question… Actually, two questions:

- **Q1**: Why there are not called `write()` and `query()` ?
- **Q2**: Where is the `read()` ?

A1: There are - the `write_str() / write()` and `query_str() / query()` are aliases. We promote the `_str` names, to clearly show you want to work with strings. Strings in Python3 are Unicode, the *bytes* and *string* objects are not interchangeable, since one character might be represented by more than 1 byte. To avoid mixing string and binary communication, all the method names for binary transfer contain `_bin` in the name.

A2: Short answer - you do not need it. Long answer - your instrument never sends unsolicited responses. If you send a set-command, you use `write_str()`. For a query-command, you use `query_str()`. 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 abovementioned 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 *
```

(continues on next page)

(continued from previous page)

```
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_str("INIT")
instr.query_opc()

# The results are ready now to fetch
results = instr.query_str('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_str("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

EIGHT

6. 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:

```
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:

```
errors_list = instr.query_all_errors()
```

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

CHAPTER

NINE

7. 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 *

instr = None
# Try-catch for initialization. If an error occurs, the ResourceError is raised
try:
    instr = RsInstrument('TCPIP::10.112.1.179::HISLIP', True, True)
except ResourceError as e:
    print(e.args[0])
    print('Your instrument is probably OFF...')
    # Exit now, no point of continuing
    exit(1)

# Dealing with commands that potentially generate errors OPTION 1:
# Switching the status checking OFF temporarily
instr.instrument_status_checking = False
instr.write_str('MY:MISSpelled:COMMand')
# Clear the error queue
instr.clear_status()
# Status checking ON again
instr.instrument_status_checking = True

# Dealing with queries that potentially generate errors OPTION 2:
try:
    # You might want to reduce the VISA timeout to avoid long waiting
    instr.visa_timeout = 1000
    instr.query_str('MY:OTHEr:WRONg:QUERy?')
```

(continues on next page)

(continued from previous page)

```
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.visa_timeout = 5000
    # Close the session in any case
    instr.close()
```

Tip: General rules for exception handling:

- If you are sending commands that might generate errors in the instrument, for example deleting a file which does not exist, use the **OPTION 1** - temporarily disable status checking, send the command, clear the error queue and enable the status checking again.
- If you are sending queries that might generate errors or timeouts, for example querying measurement that cannot be performed at the moment, use the **OPTION 2** - try/except with optionally adjusting timeouts.

CHAPTER

TEN

8. 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_str()` has `write_str_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)

```
# READ command starts the measurement, we use query_with_opc to wait for the measurement␣
↪to finish
result2 = instr.query_str_with_opc('READ:MEASUREMENT?', 50000)

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

CHAPTER

ELEVEN

9. 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.

11.1 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 endianness

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

11.2 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)

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


# Query array of integers in Binary format
t = time()
# This tells the RsInstrument in which format to expect the binary integer data
rto.bin_int_numbers_format = BinIntFormat.Integer32_4bytes
# If your instrument sends the data with the swapped endianness, use the following␣
↪format:
# rto.bin_int_numbers_format = BinIntFormat.Integer32_4bytes_swapped
waveform = rto.query_bin_or_ascii_int_list('FORM INT,32;:CHAN1:DATA?')
print(f'Instrument returned {len(waveform)} points in binary format, query duration
↪{time() - t:.3f} secs')

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

CHAPTER

TWELVE

10. 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: 12_Transferring_Files.

12.1 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?')
```

12.2 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

THIRTEEN

11. WRITING BINARY DATA

13.1 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

13.2 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

FOURTEEN

12. TRANSFERRING FILES

14.1 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')
```

14.2 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')
```

CHAPTER

FIFTEEN

13. 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 (programmers fancy name is `handler` or `callback`), which is then periodicaly 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

SIXTEEN

14. 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:

16.1 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?')


# Make sure you have the RsInstrument version 1.9.0 and newer
RsInstrument.assert_minimum_version('1.9.0')
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')
```

(continues on next page)

(continued from previous page)

```
instr.close()
```

16.2 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')


# Make sure you have the RsInstrument version 1.9.0 and newer
RsInstrument.assert_minimum_version('1.9.0')
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')
```

(continues on next page)

(continued from previous page)

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

As you see, everything works fine. If you want to simulate some party crosstalk, uncomment the line `instr2.clear_lock()`. Thich 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.

16.3 Multiple instrument sessions accessed from multiple threads

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

- 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')


# Make sure you have the RsInstrument version 1.9.0 and newer
RsInstrument.assert_minimum_version('1.9.0')
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)
```

(continues on next page)

(continued from previous page)

```
    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

SEVENTEEN

15. 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 more):

- **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** (coming in the near future).

Let us take this basic example:

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

from RsInstrument import *

# Make sure you have the RsInstrument version 1.14.0 and newer
RsInstrument.assert_minimum_version('1.14.0')
instr = RsInstrument('TCPIP::192.168.1.101::INSTR')

# Switch ON logging to the console.
instr.logger.log_to_console = True
instr.logger.mode = LoggingMode.On
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 the 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::HISLIP', options='LoggingMode=On')
```

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 *

# Make sure you have the RsInstrument version 1.14.0 and newer
RsInstrument.assert_minimum_version('1.14.0')
instr = RsInstrument('TCPIP::192.168.1.101::INSTR')

# 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.mode = LoggingMode.On

# Instead of the 'TCPIP::192.168.1.101::INSTR', show 'MyDevice'
```

(continues on next page)

(continued from previous page)

```
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()
```

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 *

# Make sure you have the RsInstrument version 1.14.0 and newer
RsInstrument.assert_minimum_version('1.14.0')

# log file common for all the instruments
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
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 string: *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
```

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
```

Another cool feature is logging only errors. To make this mode usefull for troubleshooting, you also want to see the circumstances which lead to the errors. Each RsInstrument elementary operation, for example, `write_str()`, 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 *

# Make sure you have the RsInstrument version 1.14.0 and newer
RsInstrument.assert_minimum_version('1.14.0')
instr = RsInstrument('TCPIP::192.168.1.101::INSTR', options='LoggingMode=Errors')

# Switch ON logging to the console.
```

(continues on next page)

(continued from previous page)

```
instr.logger.log_to_console = True

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

# 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 string: *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 string: *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 went error-free.

CHAPTER

EIGHTEEN

RSINSTRUMENT PACKAGE

18.1 Subpackages

18.2 Submodules

18.3 RsInstrument.RsInstrument module

Root class for remote-controlling instrument with SCPI commands.

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

Bases: `object`

Root class for remote-controlling instrument with SCPI commands.

Initializes new RsInstrument session.

Parameter options tokens examples:

- `Simulate=True` - starts the session in simulation mode. Default: `False`
- `SelectVisa=socket` - 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
- `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
- `OpcWaitMode = OpcQuery` - mode for all the opc-synchronised write/reads. Other modes: StbPolling, StbPollingSlow, StbPollingSuperSlow. Default: `StbPolling`
- `AddTermCharToWriteBinBLock = True` - Adds one additional LF to the end of the binary data (some instruments require that). Default: `False`
- `AssureWriteWithTermChar = True` - Makes sure each command/query is terminated with termination character. Default: Interface dependent
- `TerminationCharacter = "\r"` - Sets the termination character for reading. Default: `\n` (LineFeed or LF)

- `DataChunkSize = 10E3` - Maximum size of one write/read segment. If transferred data is bigger, it is split to more segments. Default: `1E6` 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`
- `StbInErrorCheck = False` - if true, the driver checks errors with *STB? If false, it uses SYST:ERR?. Default: `True`
- `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>`
- `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
- `Encoding = "utf-8", Setting of encoding for strings into bytes and vice-versa. Default: ``charmap`
- `Profile = "hm8123", Setting profile fitting the specific non-standard instruments. Default: ``none`

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
- **options** – string tokens alternating the driver settings
- **direct_session** – Another driver object or pyVisa object to reuse the session instead of opening a new session

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: threading.RLock*) → None

Assigns the provided thread lock.

property bin_float_numbers_format: RsInstrument.Internal.Conversions.BinFloatFormat

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

property bin_int_numbers_format: RsInstrument.Internal.Conversions.BinIntFormat

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

clear_lock()

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 instrument driver version

property encoding: str

Returns string<=>bytes encoding of the session.

property events: *RsInstrument.Fixed_Files.Events.Events*

Interface for event handlers, see *here*

classmethod from_existing_session(*session: object*, *options: Optional[str] = None*) → *RsInstrument.RsInstrument.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.

property full_instrument_model_name: str

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

get_last_sent_cmd() → str

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

get_lock() → threading.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

property idn_string: str

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

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.

static list_resources(*expression: str = '?*::INSTR'*, *visa_select: Optional[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'

property logger: *RsInstrument.Internal.ScpiLogger.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_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]

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]]

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: Optional[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: Optional[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 in ASCII format or in 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: Optional[int] = None*) → List[float]

Sends a OPC-synced query and reads an list of floating-point numbers that can be returned in ASCII format or in 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 in ASCII format or in 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: Optional[int] = None*) → List[int]

Sends a OPC-synced query and reads an list of floating-point numbers that can be returned in ASCII format or in 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 (',').

query_bool_list_with_opc(*query: str*, *timeout: Optional[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.

query_bool_with_opc(*query: str*, *timeout: Optional[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: Optional[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: Optional[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*) → 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.

query_str_list_with_opc(*query: str*, *timeout: Optional[int] = None*) → 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.

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.

query_str_with_opc(*query: str*, *timeout: Optional[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_with_opc(*query: str*, *timeout: Optional[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.

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.

property resource_name: str

Returns the resource name used in the constructor

self_test(*timeout: Optional[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.

property supported_models: List[str]

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

property visa_manufacturer: int

Returns the manufacturer of the current VISA session.

property visa_timeout: int

Sets / returns visa IO timeout in milliseconds.

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: Optional[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: Optional[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: Optional[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 the write() method.

write_str_with_opc(*cmd: str*, *timeout: Optional[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: Optional[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.

18.4 Module contents

VISA communication interface for SCPI-based instrument remote control. :version: 1.22.0.80 :copyright: 2020 by Rohde & Schwarz GMBH & Co. KG :license: MIT, see LICENSE for more details.

CHAPTER

NINETEEN

RSINSTRUMENT.LOGGER

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

class ScpiLogger

Base class for SCPI logging

mode

Sets / returns the Logging mode.

Data Type LoggingMode

default_mode

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

Data Type LoggingMode

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: Optional[bool] = None*) → None

Sets logging target - the target must implement write() and flush(). You can optionally set the console logging ON or OFF.

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.datetime*, *end_time: datetime.datetime*, *log_string_info: str*, *log_string: str*) → None

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

error(*start_time: datetime.datetime*, *end_time: datetime.datetime*, *log_string_info: str*, *log_string: str*) → None

Method for logging one error entry.

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.

Data Type boolean

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%`

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.

CHAPTER

TWENTY

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`

CHAPTER

TWENTYONE

INDEX AND SEARCH

- genindex
- search

PYTHON MODULE INDEX

r

INDEX

P

Q

R

S

U

V

W