RsOsp

Release 2.10.17.75

Rohde & Schwarz

Jul 12, 2021

CONTENTS:

CHAPTER

ONE

GETTING STARTED

1.1 Introduction

RsOsp is a Python remote-control communication module for Rohde & Schwarz SCPI-based Test and Measurement Instruments. It represents SCPI commands as fixed APIs and hence provides SCPI autocompletion and helps you to avoid common string typing mistakes.

> Basic example of the idea:
>
> SCPI command:
>
> `SYSTem:REFerence:FREQuency:SOURce`
>
> Python module representation:
>
> writing:
>
> `driver.system.reference.frequency.source.set()`
>
> reading:
>
> `driver.system.reference.frequency.source.get()`

Check out this RsOsp example:

```
import time

from RsOsp import *

RsOsp.assert_minimum_version('2.10')
osp = RsOsp(f'TCPIP::10.212.0.85::INSTR')

osp.utilities.visa_timeout = 5000
# Sends OPC after each commands
osp.utilities.opc_query_after_write = True

osp.utilities.reset()

# Self-test
self_test = osp.utilities.self_test()
print(f'Hello, I am {osp.utilities.idn_string}\n')
```

(continues on next page)

(continued from previous page)

```
osp.route.path.delete_all()
osp.route.path.define.set("Test1", "(@F01M01(0201, 0302))")
paths2 = osp.route.path.get_catalog()

print(f'Osp defined paths:\n {",".join(osp.route.path.get_catalog())}')
path_last = osp.route.path.get_last()
path_list = osp.route.path.get_catalog()
pathname = path_list[0]
print(f'Defined Path Definitions: {len(path_list)}')
for pathname in path_list:
    print(f' Path Name:  {pathname} ({osp.route.path.define.get(pathname)} )')
    osp.route.close.set_path(pathname)
    time.sleep(1)

print(f'Osp errors\n:{osp.utilities.query_all_errors()}')

osp.close()
```

Couple of reasons why to choose this module over plain SCPI approach:

- Type-safe API using typing module
- You can still use the plain SCPI communication
- 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 (for big files transfer)
- Multithreading session locking - you can use multiple threads talking to one instrument at the same time
- Logging feature tailored for SCPI communication - different for binary and ascii data

1.2 Installation

RsOsp 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`.

Preconditions

- Installed VISA. You can skip this if you plan to use only socket LAN connection. Download the Rohde & Schwarz VISA for Windows, Linux, Mac OS from here

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\Python37\Scripts
  ```

- Install with the command: `pip install RsOsp`

Option 2 - Installing in Pycharm

- In Pycharm Menu `File->Settings->Project->Project Interpreter` click on the '+' button on the top left (the last PyCharm version)
- Type `RsOsp` in the search box
- 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_Web_Series .

Option 3 - Offline Installation

If you are still reading the installation chapter, it is probably because the options above did not work for you - proxy problems, your boss saw the internet bill… Here are 5 easy step for installing the RsOsp offline:

- Download this python script (**Save target as**): rsinstrument_offline_install.py This installs all the preconditions that the RsOsp needs.
- Execute the script in your offline computer (supported is python 3.6 or newer)
- Download the RsOsp package to your computer from the pypi.org: https://pypi.org/project/RsOsp/#files to for example `c:\temp\`
- Start the command line `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\Python37\Scripts
  ```

- Install with the command: `pip install c:\temp\RsOsp-2.10.17.75.tar`

1.3 Finding Available Instruments

Like the pyvisa's ResourceManager, the RsOsp can search for available instruments:

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

from RsOsp import *

# Use the instr_list string items as resource names in the RsOsp constructor
instr_list = RsOsp.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 RsOsp import *

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

Tip: We believe our R&S VISA is the best choice for our customers. Here are the reasons why:

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

1.4 Initiating Instrument Session

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

Standard Session Initialization

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

```
"""
Simple example on how to use the RsOsp module for remote-controlling your instrument
Preconditions:

- Installed RsOsp Python module Version 2.10.17 or newer from pypi.org
- Installed VISA, for example R&S Visa 5.12 or newer
"""

from RsOsp import *

# A good practice is to assure that you have a certain minimum version installed
RsOsp.assert_minimum_version('2.10.17')
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)

# Initializing the session
driver = RsOsp(resource_string_1)

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

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

Note: If you are wondering about the missing `ASRL1::INSTR`, yes, it works too, but come on… it's 2021.

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

- `idn_string`
- `driver_version`
- `visa_manufacturer`
- `full_instrument_model_name`
- `instrument_serial_number`
- `instrument_firmware_version`
- `instrument_options`

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

```
driver = RsOsp('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 RsOsp module.
- Setting `reset` to True (default is False) resets your instrument. It is equivalent to calling the `reset()` method.

Selecting a Specific VISA

Just like in the function `list_resources()`, the RsOsp allows you to choose which VISA to use:

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

from RsOsp import *

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

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

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

No VISA Session

We recommend using VISA when possible preferrably with HiSlip session because of its low latency. However, if you are a strict VISA denier, RsOsp has something for you too - **no Visa installation raw LAN socket**:

```
"""
Using RsOsp without VISA for LAN Raw socket communication
"""

from RsOsp import *

driver = RsOsp('TCPIP::192.168.56.101::5025::SOCKET', True, True, "SelectVisa='socket'")
print(f'Visa manufacturer: {driver.utilities.visa_manufacturer}')
print(f"\nHello, I am: '{driver.utilities.idn_string}'")

# Close the session
driver.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.

Simulating Session

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

```
driver = RsOsp('TCPIP::192.168.56.101::HISLIP', True, True, "Simulate=True")
```

More `option_string` tokens are separated by comma:

```
driver = RsOsp('TCPIP::192.168.56.101::HISLIP', True, True, "SelectVisa='rs',␣
↪Simulate=True")
```

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 RsOsp objects:

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

from RsOsp import *

driver1 = RsOsp('TCPIP::192.168.56.101::INSTR', True, True)
driver2 = RsOsp.from_existing_session(driver1)

print(f'driver1: {driver1.utilities.idn_string}')
print(f'driver2: {driver2.utilities.idn_string}')

# Closing the driver2 session does not close the driver1 session - driver1 is the
↪'session master'
driver2.close()
print(f'driver2: I am closed now')

print(f'driver1: I am  still opened and working: {driver1.utilities.idn_string}')
driver1.close()
print(f'driver1: Only now I am closed.')
```

Note: The `driver1` is the object holding the 'master' session. If you call the `driver1.close()`, the `driver2` loses its instrument session as well, and becomes pretty much useless.

1.5 Plain SCPI Communication

After you have opened the session, you can use the instrument-specific part described in the RsOsp API Structure. If for any reason you want to use the plain SCPI, use the `utilities` interface's two basic methods:

- `write_str()` - writing a command without an answer, for example ***RST**
- `query_str()` - querying your instrument, for example the ***IDN?** query

You may ask a question. Actually, two questions:

- **Q1**: Why there are not called `write()` and `query()` ?

- **Q2**: Where is the `read()` ?

Answer 1: Actually, there are - the `write_str()` / `write()` and `query_str()` / `query()` are aliases, and you can use any of them. 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.

Answer 2: 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…

Bottom line - if you are used to `write()` and `query()` methods, from pyvisa, the `write_str()` and `query_str()` are their equivalents.

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

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

from RsOsp import *

driver = RsOsp('TCPIP::192.168.56.101::INSTR')
driver.utilities.write_str('*RST')
response = driver.utilities.query_str('*IDN?')
print(response)

# Close the session
driver.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 RsOsp import *

driver = RsOsp('TCPIP::192.168.56.101::INSTR')
driver.utilities.reset()
print(driver.utilities.idn_string)

# Close the session
driver.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
driver.utilities.visa_timeout = 3000
```

After this time, the RsOsp raises an exception. Speaking of exceptions, an important feature of the RsOsp 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 RsOsp import *

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

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

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

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

```
driver.utilities.visa_timeout = 3000
driver.utilities.write_str("INIT")
driver.utilities.query_opc()

# The results are ready now to fetch
results = driver.utilities.query_str("FETCH:MEASUREMENT?")
```

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

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

1.6 Error Checking

RsOsp pushes limits even further (internal R&S joke): It 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
driver.utilities.instrument_status_checking = False
```

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

```
driver.utilities.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 this snippet:

```
errors_list = driver.utilities.query_all_errors()
```

See the next chapter on how to react on errors.

1.7 Exception Handling

The base class for all the exceptions raised by the RsOsp 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. Because it is difficult to generate an error using the instrument-specific SCPI API, we use plain SCPI commands:

```
"""
Showing how to deal with exceptions
"""

from RsOsp import *

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

(continues on next page)

(continued from previous page)

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

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

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 RsOsp exceptions
    print(e.args[0])
    print('Some other RsOsp error...')

finally:
    driver.utilities.visa_timeout = 5000
    # Close the session in any case
    driver.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 can not be performed at the moment, use the **OPTION 2** - try/except with optionally adjusting the timeouts.

1.8 Transferring Files

Instrument -> PC

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

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

PC -> Instrument

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

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

1.9 Writing Binary Data

Writing from bytes

An example where you need to send binary data is a waveform file of a vector signal generator. First, you compose your wform_data as `bytes`, and then you send it with `write_bin_block()`:

```
# MyWaveform.wv is an instrument file name under which this data is stored
driver.utilities.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 binary data to send

Writing from PC files

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

```
driver.utilities.write_bin_block_from_file(
    "SOUR:BB:ARB:WAV:DATA 'MyWaveform.wv',",
    r"c:\temp\wform_data.wv")
```

1.10 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, the RsOsp has this covered. And, this feature is quite universal - not just for big files transfer, but for any data in both directions.

RsOsp allows you to register a function (programmers fancy name is `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:

```python
"""
Event handlers by reading
"""

from RsOsp 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)


driver = RsOsp('TCPIP::192.168.56.101::INSTR')

driver.events.on_read_handler = my_transfer_handler
# Switch on the data to be included in the event arguments
# The event arguments args.data will be updated
driver.events.io_events_include_data = True
# Set data chunk size to 2 bytes
driver.utilities.data_chunk_size = 2
driver.utilities.query_str('*IDN?')
# Unregister the event handler
driver.utilities.on_read_handler = None

# Close the session
driver.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 RsOsp 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, 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:

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

1.11 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, RsOsp 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 mulithread scenarios:

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 RsOsp object
"""

import threading
from RsOsp import *


def execute(session):
    """Executed in a separate thread."""
    session.utilities.query_str('*IDN?')


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

```

(continues on next page)

(continued from previous page)

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

driver.close()
```

Shared instrument session, accessed from multiple threads

Same as 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 RsOsp objects with shared session
"""

import threading
from RsOsp import *


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


driver1 = RsOsp('TCPIP::192.168.56.101::INSTR')
driver2 = RsOsp.from_existing_session(driver1)
driver1.utilities.visa_timeout = 200
driver2.utilities.visa_timeout = 200
# To see the effect of crosstalk, uncomment this line
# driver2.utilities.clear_lock()

threads = []
for i in range(10):
    t = threading.Thread(target=execute, args=(driver1, 1, i,))
    t.start()
    threads.append(t)
    t = threading.Thread(target=execute, args=(driver2, 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)

```
driver2.close()
driver1.close()
```

As you see, everything works fine. If you want to simulate some party crosstalk, uncomment the line `driver2.utilities.clear_lock()`. Thich causes the driver2 session lock to break away from the driver1 session lock. Although the driver1 still tries to schedule its instrument access, the driver2 tries to do the same at the same time, which leads to all the fun stuff happening.

Multiple instrument sessions accessed from multiple threads

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

- Your 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 RsOsp takes care of it for you. The text below describes this scenario.

Run the following example:

```
"""
Multiple threads are accessing two RsOsp objects with two separate sessions
"""

import threading
from RsOsp import *


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


driver1 = RsOsp('TCPIP::192.168.56.101::INSTR')
driver2 = RsOsp('TCPIP::192.168.56.101::INSTR')
driver1.utilities.visa_timeout = 200
driver2.utilities.visa_timeout = 200

# Synchronise the sessions by sharing the same lock
driver2.utilities.assign_lock(driver1.utilities.get_lock())  # To see the effect of␣
↪crosstalk, comment this line

threads = []
for i in range(10):
    t = threading.Thread(target=execute, args=(driver1, 1, i,))
    t.start()
    threads.append(t)
    t = threading.Thread(target=execute, args=(driver2, 2, i,))
    t.start()
```

(continues on next page)

(continued from previous page)

```
    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')

driver2.close()
driver1.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 `driver2.utilities.assign_lock(driver1.utilities.get_lock())` Try to comment it 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.

1.12 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 actually 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

Let us take this basic example:

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

from RsOsp import *

driver = RsOsp('TCPIP::192.168.1.101::INSTR')

# Switch ON logging to the console.
driver.utilities.logger.log_to_console = True
driver.utilities.logger.mode = LoggingMode.On
driver.utilities.reset()

# Close the session
driver.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
```

(continues on next page)

(continued from previous page)

```
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 `driver.reset()`. If you want to log the initialization of the session as well, you have to switch the logging ON already in the constructor:

```
driver = RsOsp('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 RsOsp import *

driver = RsOsp('TCPIP::192.168.1.101::INSTR')

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

# Logging target is our file
file = open(r'c:\temp\my_file.txt', 'w')
driver.utilities.logger.set_logging_target(file)
driver.utilities.logger.mode = LoggingMode.On

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

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

(continues on next page)

(continued from previous page)

```
driver.utilities.reset()

# Close the session
driver.close()

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

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

```
driver.utilities.logger.log_status_check_ok = False
```

Hint: You can share the logging file between multiple sessions. In such case, remember to close the file only after you have stopped logging in all your sessions, otherwise you get a log write error.

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 driver 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 use a direct SCPI communication to send a misspelled SCPI command *CLS, which leads to instrument status error:

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

from RsOsp import *

driver = RsOsp('TCPIP::192.168.1.101::INSTR', options='LoggingMode=Errors')

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

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

# Now a misspelled command.
driver.utilities.write('*CLaS')

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

# Close the session
driver.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 were all error-free.

CHAPTER TWO

REVISION HISTORY

Rohde & Schwarz OSP Opens Switch Platform RsOsp instrument driver.

Supported instruments: OSP

The package is hosted here: https://pypi.org/project/RsOsp/

Documentation: https://RsOsp.readthedocs.io/

Examples: https://github.com/Rohde-Schwarz/Examples/tree/main/Misc/Python/RsOsp_ScpiPackage

---

Release Notes:

Latest release notes summary: Release for new OSP FW 2.10.17

Version 2.10.17.75

- Release for new OSP FW 2.10.17

Version 1.0.4.57

- Added documentation on ReadTheDocs

Version 1.0.4.57

- Fixed formatting of string in methods with list of paths as inputs

Version 1.0.3.55

- Changed responses for methods with List[string] return values: If the instrument returns exactly one empty string, the methods return empty List []

Version 1.0.2.54

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

Version 1.0.0.50

- First released version

CHAPTER

THREE

ENUMS

3.1 ReplaceOrKeep

```
# Example value:
value = enums.ReplaceOrKeep.KEEP
# All values (2x):
KEEP | REPLace
```

3.2 TriggerExecType

```
# Example value:
value = enums.TriggerExecType.RESet
# All values (2x):
RESet | TRIGger
```

3.3 TriggerSlope

```
# Example value:
value = enums.TriggerSlope.BOTH
# All values (3x):
BOTH | NEGative | POSitive
```

3.4 TriggerType

```
# Example value:
value = enums.TriggerType.ADDRessed
# All values (4x):
ADDRessed | SEQuenced | SINGle | TOGGle
```

CHAPTER

FOUR

EXAMPLES

For more examples, visit our Rohde & Schwarz Github repository.

```
import time

from RsOsp import *

RsOsp.assert_minimum_version('2.10')
osp = RsOsp(f'TCPIP::10.212.0.85::INSTR')

osp.utilities.visa_timeout = 5000
# Sends OPC after each commands
osp.utilities.opc_query_after_write = True

osp.utilities.reset()

# Self-test
self_test = osp.utilities.self_test()
print(f'Hello, I am {osp.utilities.idn_string}\n')

osp.route.path.delete_all()
osp.route.path.define.set("Test1", "(@F01M01(0201, 0302))")
paths2 = osp.route.path.get_catalog()

print(f'Osp defined paths:\n {",".join(osp.route.path.get_catalog())}')
path_last = osp.route.path.get_last()
path_list = osp.route.path.get_catalog()
pathname = path_list[0]
print(f'Defined Path Definitions: {len(path_list)}')
for pathname in path_list:
    print(f' Path Name:  {pathname} ({osp.route.path.define.get(pathname)} )')
    osp.route.close.set_path(pathname)
    time.sleep(1)

print(f'Osp errors\n:{osp.utilities.query_all_errors()}')

osp.close()
```

```
import time

from RsOsp import *
```

(continues on next page)

(continued from previous page)

```
def ask(prompt, typ, default):
    print('%s [%s] ' % (prompt, default)),
    value = typ(input())
    if not value:
        return default
    else:
        return value


RsOsp.assert_minimum_version('2.10')
ip = ask('Enter IP address of OSP: ', str, '10.212.0.85')

print('-------------------------------------------------------------------------')
print('---    O S P   D E V I C E   I N F O R M A T I O N      -----------------')
print('-------------------------------------------------------------------------')
osp_base = RsOsp(f'TCPIP::{ip}::INSTR', True, False)

osp_base.utilities.visa_timeout = 5000
# Sends OPC after each commands
osp_base.utilities.opc_query_after_write = True
# Checks for syst:err? after each command / query
osp_base.utilities.instrument_status_checking = True

# You can still use the direct SCPI write / query:
osp_base.utilities.write_str('*RST')
instr_err = osp_base.utilities.query_str('SYST:ERR?')
# System Reset
osp_base.utilities.reset()

# Self-test
self_test = osp_base.utilities.self_test()

print(f'Identification ...............=: {osp_base.utilities.idn_string}\n')
print(f'Instrument Manufacturer.......=: {osp_base.utilities.manufacturer}')
print(f'Instrument Name        .......=: {osp_base.utilities.full_instrument_model_name}
↪')
print(f'Instrument Serial Number......=: {osp_base.utilities.instrument_serial_number}')
print(f'Instrument Firmware Version ..=: {osp_base.utilities.instrument_firmware_version}
↪')
print(f'Instrument Options...........=: {",".join(osp_base.utilities.instrument_
↪options)}\n')
print(f'Supported Devices..    .......=: {",".join(osp_base.utilities.supported_models)}
↪')
print(f'VISA Manufacturer............=: {osp_base.utilities.visa_manufacturer}')
print(f'VISA Timeout.................=: {osp_base.utilities.visa_timeout}')
print(f'Driver Version ..............=: {osp_base.utilities.driver_version}\n')

#    print(f'Osp instrument status:{osp_base.utilities.instrument_status_checking}')

print(f'Osp HwInfo:\n{",".join(osp_base.diagnostic.service.get_hw_info())}')
```

(continues on next page)

(continued from previous page)

```
print(f'Osp virtual mode enable ?:{osp_base.configure.virtual.get_mode()}')
if osp_base.configure.virtual.get_mode() is False:
    osp_base.configure.virtual.set_mode(True)
print(f'Osp virtual mode enable ?:{osp_base.configure.virtual.get_mode()}')

osp_base.route.path.delete_all()
paths = osp_base.route.path.get_catalog()

osp_base.route.path.define.set("Test1", "(@F01M01(0201, 0302))")
paths2 = osp_base.route.path.get_catalog()

print(f'Osp HwInfo:\n{",".join(osp_base.diagnostic.service.get_hw_info())}')
hwinfolist = osp_base.diagnostic.service.get_hw_info()
print(hwinfolist)

for hw in hwinfolist:
    print(hw)
    module_info = hw.rsplit("|")
    print(module_info)
    print(module_info[1])
    time.sleep(1)

# get path list and switch all after each other
print(f'Osp defined paths:\n {",".join(osp_base.route.path.get_catalog())}')
path_last = osp_base.route.path.get_last()
path_list = osp_base.route.path.get_catalog()
path_name = path_list[0]
print(f'Defined Path Definitions: {len(path_list)}')
for path_name in path_list:
    print(f' Path Name:  {path_name} ({osp_base.route.path.define.get(path_name)} )')
    osp_base.route.close.set_path(path_name)
    print(f'Osp error?:{osp_base.utilities.query_str("SYST:ERR?")}')
    time.sleep(1)

print(f'Osp error?:{osp_base.utilities.query_str("SYST:ERR?")}')
osp_base.utilities.reset()

osp_base.configure.virtual.set_mode(False)

osp_base.close()
```

CHAPTER

FIVE

RSOSP API STRUCTURE

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

63 total commands, 5 Sub-groups, 0 group commands

Initializes new RsOsp 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=ivi` - 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 = LoggingMode.On` - Sets the logging status right from the start. Default: `Off`
- `LoggingName = 'MyDevice'` - Sets the name to represent the session in the log entries. Default: `'resource_name'`

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 sybsystem
- **`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.

`close()` → None

Closes the active RsOsp session.

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

Creates a new RsOsp object with the entered 'session' reused.

Parameters

- **`session`** – can be an another driver or a direct pyvisa session.
- **`options`** – string tokens alternating the driver settings.

`get_session_handle()` → object

Returns the underlying session handle.

`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'

Subgroups

5.1 Configure

class Configure

Configure commands group definition. 24 total commands, 8 Sub-groups, 0 group commands

Subgroups

5.1.1 Frame

SCPI Commands

```
CONFigure:FRAMe:CATalog
CONFigure:FRAMe:ADD
CONFigure:FRAMe:DELete
CONFigure:FRAMe:DELete:ALL
CONFigure:FRAMe:EXPort
```

class Frame

Frame commands group definition. 11 total commands, 3 Sub-groups, 5 group commands

delete(*frame_id: str*) → None

```
# SCPI: CONFigure:FRAMe:DELete
driver.configure.frame.delete(frame_id = r1)
```

Deletes the definition of a selected secondary switch unit from the primary switch unit's internal volatile memory. Use the command method RsOsp.Configure.Frame.catalog to query the existing secondary switch unit definitions. When you delete a secondary switch unit from an existing list of secondary devices, all following frame IDs of secondary switch units listed after the deleted device are automatically renumbered (decremented by 1) . For example, if you delete a secondary switch unit with frame ID F03, the next remaining secondary device F04 becomes secondary device F03, the next remaining secondary device F05 becomes secondary device F04, etc. Note that the deletion of a secondary switch unit can impact your path definitions, even if the deleted frame was not used for any path definitions. For example, consider a setup with 4 secondary devices (F02 to F05) . If you delete the 2nd secondary device (F03) , a path that includes modules in the previous 3rd secondary device (F04) now addresses the new 3rd secondary device, which previously was the 4th secondary device (F05) . This change can destroy the functionality of your path definitions - or it can be intentional.

param frame_id Selects the frame ID Fxx of the secondary switch unit you wish to delete, starting with F02 (note that the 1st secondary switch unit is the 2nd frame) . Use the frame ID without quotation marks.

delete_all() → None

```
# SCPI: CONFigure:FRAMe:DELete:ALL
driver.configure.frame.delete_all()
```

Deletes all currently defined secondary switch units from the primary switch unit's internal volatile memory. Before you delete all secondary switch unit definitions, we recommend using the command method RsOsp.Configure.Frame.catalog to query all currently defined secondary switch units.

delete_all_with_opc() → None

```
# SCPI: CONFigure:FRAMe:DELete:ALL
driver.configure.frame.delete_all_with_opc()
```

Deletes all currently defined secondary switch units from the primary switch unit's internal volatile memory. Before you delete all secondary switch unit definitions, we recommend using the command method RsOsp.Configure.Frame.catalog to query all currently defined secondary switch units.

Same as delete_all, but waits for the operation to complete before continuing further. Use the RsOsp.utilities.opc_timeout_set() to set the timeout value.

export(*slave_config_file: str*) → None

```
# SCPI: CONFigure:FRAMe:EXPort
driver.configure.frame.export(slave_config_file = '1')
```

Stores the currently defined secondary devices configuration as a nonvolatile file in the compact flash memory of your primary switch unit. For configuring secondary switch units, see method RsOsp.Configure.Frame.Define.set. All secondary switch unit configuration filenames have the extension '.iconn'. Do not enter the extension when specifying a filename. A filename query does not return the extension. For example, when you save the interconnection configuration file 'subunit1', it is saved as 'subunit1.iconn'. A query returns this filename as 'subunit1', only.

param slave_config_file String parameter to specify the name of the file to be stored.

get_catalog() → List[str]

```
# SCPI: CONFigure:FRAMe:CATalog
value: List[str] = driver.configure.frame.get_catalog()
```

Returns a comma-separated configuration list of all 'frames' (switch units in single state or in an interconnection setup) that are stored in the primary switch unit's volatile memory. Use the command method RsOsp.Configure.Frame.export to save the configuration list to the switch unit's compact flash memory. The configuration is also saved automatically to the flash memory during the shutdown procedure and loaded at startup into the volatile memory.

return frame_info_list: The information on each frame comprises the following information: Fxx|Address|Status|TransmittedHostname. In detail: - Fxx: Frame ID, where F01 is the primary switch unit, F02 is the 1st secondary switch unit, F03 is the 2nd secondary switch unit, and so on. - Address: IP address or configured hostname, as specified by CONF:FRAM:DEF or in the user interface ('WebGUI') at 'Configuration' 'Interconnection' 'Edit Secondary' 'Address:'No address is defined for the primary switch unit (F01) . For the primary switch unit, the query returns an empty field in the response string.For primary and secondary switch units that are in 'Virtual Mode', the query returns Not available (virtual frame) in this field. - Status: For example, Single, Primary, or in the secondary switch units: Connected, Broken (secondary switch unit not available) , Refused (when trying to configure another primary switch unit as a secondary switch unit) . - TransmittedHostname: Hostname, if available, or Not available (virtual frame) .If no address or hostname is defined for an existing frame, for example due to an incomplete definition in the user interface ('WebGUI') , the query returns an empty field.For example, the response can be 'F02||Invalid address|'.

set_add(*configured_address: str*) → None

```
# SCPI: CONFigure:FRAMe:ADD
driver.configure.frame.set_add(configured_address = '1')
```

Adds an entry for a secondary switch unit at the end of the list of frame IDs in the switch unit's internal volatile memory. The command assigns the next available frame ID to the new secondary switch unit.

param configured_address Specifies the IP address or the hostname of the secondary switch unit that you want to add.

Subgroups

5.1.1.1 Define

SCPI Commands

```
CONFigure:FRAMe:DEFine
```

class Define

Define commands group definition. 1 total commands, 0 Sub-groups, 1 group commands

get(*frame_id: str*) → str

```
# SCPI: CONFigure:FRAMe[:DEFine]
value: str = driver.configure.frame.define.get(frame_id = r1)
```

Defines how an existing secondary switch unit is addressed via LAN by the primary switch unit. To do so, the command selects a secondary switch unit by its IP address or hostname and allows you to set the frame ID of this secondary switch unit in the 'Interconnection' configuration of the primary switch unit. Note that this command does not change the network settings or IP address of the primary or secondary device. It only defines the ID, by which a primary device addresses a secondary device. The query returns the IP address of the secondary switch unit with the specified frame ID. To query the full list of existing secondary switch units, use the command method RsOsp.Configure.Frame.catalog. Note that you can add or insert new secondary switch units with separate commands. The current configuration is always saved automatically to the flash memory during the shutdown procedure and loaded at startup into the volatile memory. To save a specific configuration of all frames to a separate file on the switch unit's compact flash memory, use the command method RsOsp.Configure.Frame.export.

param frame_id Selects the frame ID Fxx of the secondary switch unit that you wish to modify, starting with F02 (note that the 1st secondary switch unit is the 2nd frame) . Use the frame ID without quotation marks. In a setting, if you use F01 or a frame ID, for which no secondary switch unit is defined, a SCPI error is generated. In a query, if you use F01, an empty response is returned. If you use a frame ID, for which no secondary switch unit is defined, a SCPI error is generated.

return configured_address: Specifies the IP address or hostname, at which the secondary switch unit (selected by the frameId) is available via LAN. Use the address or hostname in quotation marks.

set(*frame_id: str, configured_address: str*) → None

```
# SCPI: CONFigure:FRAMe[:DEFine]
driver.configure.frame.define.set(frame_id = r1, configured_address = '1')
```

Defines how an existing secondary switch unit is addressed via LAN by the primary switch unit. To do so, the command selects a secondary switch unit by its IP address or hostname and allows you to set the frame ID of this secondary switch unit in the 'Interconnection' configuration of the primary switch unit. Note that this command does not change the network settings or IP address of the primary or secondary device. It only defines the ID, by which a primary device addresses a secondary device. The query returns the IP address of the secondary switch unit with the specified frame ID. To query the full list of existing secondary switch units, use the command method RsOsp.Configure.Frame.catalog. Note that you can add or insert new secondary switch units with separate commands. The current configuration is always saved automatically to the flash memory during the shutdown procedure and loaded at startup into the volatile memory. To save a specific configuration of all frames to a separate file on the switch unit's compact flash memory, use the command method RsOsp.Configure.Frame.export.

param frame_id Selects the frame ID Fxx of the secondary switch unit that you wish to modify, starting with F02 (note that the 1st secondary switch unit is the 2nd frame) . Use the frame ID without quotation marks. In a setting, if you use F01 or a frame ID, for which no secondary switch unit is defined, a SCPI error is generated. In a query, if you use F01, an empty response is returned. If you use a frame ID, for which no secondary switch unit is defined, a SCPI error is generated.

param configured_address Specifies the IP address or hostname, at which the secondary switch unit (selected by the frameId) is available via LAN. Use the address or hostname in quotation marks.

5.1.1.2 Insert

SCPI Commands

```
CONFigure:FRAMe:INSert
```

class Insert

Insert commands group definition. 1 total commands, 0 Sub-groups, 1 group commands

set(*frame_id: str*, *configured_address: str*) → None

```
# SCPI: CONFigure:FRAMe:INSert
driver.configure.frame.insert.set(frame_id = r1, configured_address = '1')
```

Inserts an entry for a secondary switch unit ahead of an existing entry in the list of frame IDs in the switch unit's internal volatile memory.

param frame_id Specifies the frame ID Fxx, at which the new secondary switch unit is to be inserted. The lowest accepted frame ID is F02. Existing frame IDs from this frame ID on are automatically renumbered (incremented by 1) . If the specified frame ID is not yet defined, a SCPI error is generated.

param configured_address Specifies the IP address or the hostname of the secondary switch unit that you want to insert.

5.1.1.3 ImportPy

SCPI Commands

```
CONFigure:FRAMe:IMPort:DELete
CONFigure:FRAMe:IMPort:DELete:ALL
CONFigure:FRAMe:IMPort
```

class ImportPy

ImportPy commands group definition. 4 total commands, 1 Sub-groups, 3 group commands

delete(*slave_config_file: str*) → None

```
# SCPI: CONFigure:FRAMe:IMPort:DELete
driver.configure.frame.importPy.delete(slave_config_file = '1')
```

Risk of losing settings: Removes the specified interconnection configuration file from the primary switch unit's compact flash memory. All secondary switch unit configuration filenames have the extension '.iconn'. Do not enter the extension when specifying a filename. A filename query does not return the extension. For example, when you save the interconnection configuration file 'subunit1', it is saved as 'subunit1.iconn'. A query returns this filename as 'subunit1', only. Legacy file extensions are still supported.

param slave_config_file String parameter to specify the name of the file to be deleted. If this file does not exist, a SCPI error is generated. You can query the error with SYST:ERR?. The result can be, for example: -200,'Execution error;File does not exist.,CONF:FRAM:IMP:DEL ''setup3frameconfigs''

delete_all(*path_information: Optional[str] = None*) → None

```
# SCPI: CONFigure:FRAMe:IMPort:DELete:ALL
driver.configure.frame.importPy.delete_all(path_information = '1')
```

Risk of losing settings: Removes all interconnection configuration files from the primary switch unit's compact flash memory. Before you delete all secondary switch unit configuration files, we recommend using the command method RsOsp. **Configure.Frame.ImportPy.Catalog.get_** to query all currently defined interconnection configuration files. All secondary switch unit configuration filenames have the extension '.iconn'. Do not enter the extension when specifying a filename. A filename query does not return the extension. For example, when you save the interconnection configuration file 'subunit1', it is saved as 'subunit1.iconn'. A query returns this filename as 'subunit1', only. Legacy file extensions are still supported.

param path_information No help available

set_value(*slave_config_file: str*) → None

```
# SCPI: CONFigure:FRAMe:IMPort
driver.configure.frame.importPy.set_value(slave_config_file = '1')
```

Loads a secondary devices configuration file from the compact flash memory of your primary switch unit into its internal volatile memory. As a prerequisite, you must have exported such a file in advance, see method RsOsp.Configure.Frame. export. All secondary switch unit configuration filenames have the extension '.iconn'. Do not enter the extension when specifying a filename. A filename query does not return the extension. For example, when you save the interconnection configuration file 'subunit1', it is saved as 'subunit1.iconn'. A query returns this filename as 'subunit1', only. Legacy file extensions are still supported. Risk of losing settings: Note that this command overwrites the secondary switch units in the current frames configuration in the primary switch unit's internal memory with the secondary switch units configuration in the loaded file. To avoid losing a current secondary switch units configuration, consider saving this configuration by method RsOsp.Configure.Frame.export, before you send the import command.

param slave_config_file String parameter to specify the name of the file to be loaded.

Subgroups

5.1.1.3.1 Catalog

SCPI Commands

```
CONFigure:FRAMe:IMPort:CATalog
```

class Catalog

Catalog commands group definition. 1 total commands, 0 Sub-groups, 1 group commands

get(*path_information: Optional[str] = None*) → List[str]

```
# SCPI: CONFigure:FRAMe:IMPort:CATalog
value: List[str] = driver.configure.frame.importPy.catalog.get(path_information␣
↪= '1')
```

Returns the names of all secondary device configuration files that are stored in the switch unit's flash memory. All secondary switch unit configuration filenames have the extension '.iconn'. Do not enter the extension when specifying a filename. A filename query does not return the extension. For example, when you save the interconnection configuration file 'subunit1', it is saved as 'subunit1.iconn'. A query returns this filename as 'subunit1', only.

param path_information No help available

return list_of_exp_slave_configs: Comma-separated list of filenames, each in quotation marks.

5.1.2 Compatible

SCPI Commands

```
CONFigure:COMPatible:MODE
```

class Compatible

Compatible commands group definition. 1 total commands, 0 Sub-groups, 1 group commands

get_mode() → bool

```
# SCPI: CONFigure:COMPatible[:MODE]
value: bool = driver.configure.compatible.get_mode()
```

Enables or disables backward compatibility of some currently available RC commands with the syntax of previous firmware versions, used for the legacy switch units of the R&S OSP1xx family: R&S OSP120, R&S OSP130 and R&S OSP150. The query returns the state of the compatibility mode. Note that both the current and the deprecated RC commands always are interpreted correctly by the firmware, independent of your compatibility settings. However, a query like 'method RsOsp. Route.Close.set' returns channel setting strings in the format 'F01M01' with method RsOsp.Configure.Compatible.mode = OFF and in the format 'F01A11' with method RsOsp.Configure.Compatible.mode = ON.

INTRO_CMD_HELP: If the compatibility mode is enabled, the following commands are also available:

- MMEM:LOAD:STATe (new: method RsOsp.Route.Path.ImportPy.value)
- MMEM:STORe:STATe (new: method RsOsp.Route.Path.export)
- ROUTe:MODule:CATalog?

Note that some commands behave differently with or without the compatibility mode enabled. For example, method RsOsp. Route.Path.Define.set as a setting accepts both syntax versions F01M01 or F01A11. But as a query, method RsOsp.Route.Path. Define.set, sent without the compatibility mode enabled, returns the current syntax. On the contrary, with compatibility mode enabled, it returns the legacy syntax, described in section method RsOsp.Route.Path.Define.set.

return state: - 1 | ON: The set of RC commands is extended as listed above for backward compatibility with R&S OSP1xx legacy switch units. - 0 | OFF: The firmware only accepts the standard set of RC commands. No additional commands are available to provide backward compatibility.

`set_mode`(*state: bool*) → None

```
# SCPI: CONFigure:COMPatible[:MODE]
driver.configure.compatible.set_mode(state = False)
```

Enables or disables backward compatibility of some currently available RC commands with the syntax of previous firmware versions, used for the legacy switch units of the R&S OSP1xx family: R&S OSP120, R&S OSP130 and R&S OSP150. The query returns the state of the compatibility mode. Note that both the current and the deprecated RC commands always are interpreted correctly by the firmware, independent of your compatibility settings. However, a query like 'method RsOsp. Route.Close.set' returns channel setting strings in the format 'F01M01' with method RsOsp.Configure.Compatible.mode = OFF and in the format 'F01A11' with method RsOsp.Configure.Compatible.mode = ON.

INTRO_CMD_HELP: If the compatibility mode is enabled, the following commands are also available:

- MMEM:LOAD:STATe (new: method RsOsp.Route.Path.ImportPy.value)
- MMEM:STORe:STATe (new: method RsOsp.Route.Path.export)
- ROUTe:MODule:CATalog?

Note that some commands behave differently with or without the compatibility mode enabled. For example, method RsOsp. Route.Path.Define.set as a setting accepts both syntax versions F01M01 or F01A11. But as a query, method RsOsp.Route.Path. Define.set, sent without the compatibility mode enabled, returns the current syntax. On the contrary, with compatibility mode enabled, it returns the legacy syntax, described in section method RsOsp.Route.Path.Define.set.

param state

- 1 | ON: The set of RC commands is extended as listed above for backward compatibility with R&S OSP1xx legacy switch units.
- 0 | OFF: The firmware only accepts the standard set of RC commands. No additional commands are available to provide backward compatibility.

5.1.3 Virtual

SCPI Commands

```
CONFigure:VIRTual:MODE
```

class Virtual

Virtual commands group definition. 1 total commands, 0 Sub-groups, 1 group commands

get_mode() → bool

```
# SCPI: CONFigure:VIRTual[:MODE]
value: bool = driver.configure.virtual.get_mode()
```

Activates or deactivates the 'Virtual Mode'. The query returns the state of the virtual mode.

return state: - 1 | ON: Activates the virtual mode. - 0 | OFF: Deactivates the virtual mode.

set_mode(*state: bool*) → None

```
# SCPI: CONFigure:VIRTual[:MODE]
driver.configure.virtual.set_mode(state = False)
```

Activates or deactivates the 'Virtual Mode'. The query returns the state of the virtual mode.

param state

- 1 | ON: Activates the virtual mode.
- 0 | OFF: Deactivates the virtual mode.

5.1.4 MainInfo

SCPI Commands

```
CONFigure:MAINinfo:TEXT
CONFigure:MAINinfo:PATH
```

class MainInfo

MainInfo commands group definition. 2 total commands, 0 Sub-groups, 2 group commands

get_path() → bool

```
# SCPI: CONFigure:MAINinfo:PATH
value: bool = driver.configure.mainInfo.get_path()
```

Enables or disables displaying the Last Switched Path information in the Main page. This RC command acts equivalent to the Path Info checkbox in the 'General' settings dialog of the user interface. The query method RsOsp.Configure.MainInfo. path returns the state of this setting. The query method RsOsp.Route.Path.last returns the information on the Last Switched Path.

return state: - 1 | ON: Displaying the Last Switched Path is enabled. - 0 | OFF: Displaying the Last Switched Path is disabled.

get_text() → str

```
# SCPI: CONFigure:MAINinfo:TEXT
value: str = driver.configure.mainInfo.get_text()
```

Specifies or queries the text displayed as Customer Text in the Main page. This RC command is equivalent to the 'Customer Text' field in the 'General' configuration dialog of the user interface.

return state: Specifies the content of the Customer Info field. Enter the content in parentheses.

set_path(*state: bool*) → None

```
# SCPI: CONFigure:MAINinfo:PATH
driver.configure.mainInfo.set_path(state = False)
```

Enables or disables displaying the Last Switched Path information in the Main page. This RC command acts equivalent to the Path Info checkbox in the 'General' settings dialog of the user interface. The query method RsOsp.Configure.MainInfo. path returns the state of this setting. The query method RsOsp.Route.Path.last returns the information on the Last Switched Path.

param state

- 1 | ON: Displaying the Last Switched Path is enabled.
- 0 | OFF: Displaying the Last Switched Path is disabled.

set_text(*state: str*) → None

```
# SCPI: CONFigure:MAINinfo:TEXT
driver.configure.mainInfo.set_text(state = '1')
```

Specifies or queries the text displayed as Customer Text in the Main page. This RC command is equivalent to the 'Customer Text' field in the 'General' configuration dialog of the user interface.

param state Specifies the content of the Customer Info field. Enter the content in parentheses.

5.1.5 PowerUp

SCPI Commands

```
CONFigure:POWerup:PATH
CONFigure:POWerup:RESet
```

class PowerUp

PowerUp commands group definition. 2 total commands, 0 Sub-groups, 2 group commands

get_path() → str

```
# SCPI: CONFigure:POWerup:PATH
value: str = driver.configure.powerUp.get_path()
```

Sets or queries the switch-on action that determines, which path (if any) is switched after booting the instrument. This RC command is equivalent to the 'Switch-On Action' in the 'General' configuration dialog of the user interface. The query returns the currently set switch-on action.

return path_name: String parameter to specify the path name (see method RsOsp.Route.Path.Define.set) of the path to be switched at power-up. If you specify a path name that does not exist, the command has no effect. If you specify an empty path name string ('') , the Switch-On Action is set to None. The switch unit does not switch any path after being booted.

`get_reset()` → bool

```
# SCPI: CONFigure:POWerup:RESet
value: bool = driver.configure.powerUp.get_reset()
```

Sets or queries the Power Up reset condition of switch modules with latching relays. This setting determines, how latching relays behave after booting the switch unit. Note that this command does NOT reset the module OSP B104, which is designed for controlling external latching relays.

INTRO_CMD_HELP: The following rules apply for identifying positions of latching SPDT relays:

- On the relays' front plates, the positions are labeled as 2 and 1, with 2 being the default position
- In the graphical user interface (GUI) , the positions are 0 and 1, with 0 being the default position
- In a remote control command (SCPI) , the positions are 00 and 01, with 00 being the default value

Hence, if you take the front-plate port labels of a latching SPDT relay, subtract 1 to get the position values that the software uses for this relay. The query returns the current reset condition.

return state: - 1 | ON: At Power Up, the switch unit handles latching relays as follows:It sets all latching SPDT relays to the default ports labeled 2, which are represented in the software by position 0.It sets all latching SPxT relays to the open state. - 0 | OFF: At Power Up, the switch unit leaves all latching relays keep their previous state.

`set_path`(*path_name: str*) → None

```
# SCPI: CONFigure:POWerup:PATH
driver.configure.powerUp.set_path(path_name = '1')
```

Sets or queries the switch-on action that determines, which path (if any) is switched after booting the instrument. This RC command is equivalent to the 'Switch-On Action' in the 'General' configuration dialog of the user interface. The query returns the currently set switch-on action.

param path_name String parameter to specify the path name (see method RsOsp.Route.Path.Define.set) of the path to be switched at power-up. If you specify a path name that does not exist, the command has no effect. If you specify an empty path name string ('') , the Switch-On Action is set to None. The switch unit does not switch any path after being booted.

`set_reset`(*state: bool*) → None

```
# SCPI: CONFigure:POWerup:RESet
driver.configure.powerUp.set_reset(state = False)
```

Sets or queries the Power Up reset condition of switch modules with latching relays. This setting determines, how latching relays behave after booting the switch unit. Note that this command does NOT reset the module OSP B104, which is designed for controlling external latching relays.

INTRO_CMD_HELP: The following rules apply for identifying positions of latching SPDT relays:

- On the relays' front plates, the positions are labeled as 2 and 1, with 2 being the default position
- In the graphical user interface (GUI) , the positions are 0 and 1, with 0 being the default position
- In a remote control command (SCPI) , the positions are 00 and 01, with 00 being the default value

Hence, if you take the front-plate port labels of a latching SPDT relay, subtract 1 to get the position values that the software uses for this relay. The query returns the current reset condition.

param state

- 1 | ON: At Power Up, the switch unit handles latching relays as follows:It sets all latching SPDT relays to the default ports labeled 2, which are represented in the software by position 0.It sets all latching SPxT relays to the open state.
- 0 | OFF: At Power Up, the switch unit leaves all latching relays keep their previous state.

5.1.6 All

SCPI Commands

```
CONFigure:ALL:BACKup
```

class All

All commands group definition. 5 total commands, 1 Sub-groups, 1 group commands

set_backup(*filename_for_backup: str*) → None

```
# SCPI: CONFigure:ALL:BACKup
driver.configure.all.set_backup(filename_for_backup = '1')


   INTRO_CMD_HELP: Saves all configuration settings to a backup file. These␣
↪settings comprise the following:

    - General configuration
    - Network settings
    - Trigger configuration (optional)
    - Interconnection configuration
    - Virtual configuration
```

All configuration filenames have the extension '.backup'. Do not enter the extension when specifying a filename. A filename query does not return the extension. For example, when you save the 'Interconnection' definition file 'settings-2018-10-25', it is saved as 'settings-2018-10-25.backup'. A query returns this filename as 'settings-2018-10-25', only.

param filename_for_backup String parameter to specify the filename for the backup.

Subgroups

5.1.6.1 Restore

SCPI Commands

```
CONFigure:ALL:RESTore:DELete
CONFigure:ALL:RESTore:DELete:ALL
CONFigure:ALL:RESTore
```

class Restore

Restore commands group definition. 4 total commands, 1 Sub-groups, 3 group commands

delete(*restore_file_to_delete: str*) → None

```
# SCPI: CONFigure:ALL:RESTore:DELete
driver.configure.all.restore.delete(restore_file_to_delete = '1')
```

Deletes a selected settings backup file from the switch unit's internal flash memory. Use the command method RsOsp. **Configure.All.Restore.Catalog.get_** to query the list of available backup files. All configuration filenames have the extension '.backup'. Do not enter the extension when specifying a filename. A filename query does not return the extension. For example, when you save the 'Interconnection' definition file 'settings-2018-10-25', it is saved as 'settings-2018-10-25.backup'. A query returns this filename as 'settings-2018-10-25', only.

param restore_file_to_delete String parameter to select the backup file to be deleted. If this file does not exist, a SCPI error is generated. You can query the error with SYST:ERR?. The result can be, for example: -200,'Execution error;File does not exist.,CONFigure:ALL:RESTore:DELete ''backup1'''

delete_all(*path_information: Optional[str] = None*) → None

```
# SCPI: CONFigure:ALL:RESTore:DELete:ALL
driver.configure.all.restore.delete_all(path_information = '1')
```

Risk of losing settings: Removes all settings backup files from the switch unit's internal memory or from a removable flash memory. Before you delete all settings backup files, we recommend using the command method RsOsp.Configure.All. **Restore.Catalog.get_** to query the currently defined settings backup files. All configuration filenames have the extension '.backup'. Do not enter the extension when specifying a filename. A filename query does not return the extension. For example, when you save the 'Interconnection' definition file 'settings-2018-10-25', it is saved as 'settings-2018-10-25.backup'. A query returns this filename as 'settings-2018-10-25', only.

param path_information No help available

set_value(*restore_file_to_restore: str*) → None

```
# SCPI: CONFigure:ALL:RESTore
driver.configure.all.restore.set_value(restore_file_to_restore = '1')
```

Loads a file previously saved as a backup of all settings (see method RsOsp.Configure.All.backup) and uses it to overwrite the current settings. All configuration filenames have the extension '.backup'. Do not enter the extension when specifying a filename. A filename query does not return the extension. For example, when you save the 'Interconnection' definition file 'settings-2018-10-25', it is saved as 'settings-2018-10-25.backup'. A query returns this filename as 'settings-2018-10-25', only. Risk of losing settings: This command overwrites all current settings in the switch unit's internal memory with the settings in the loaded file. To avoid losing current settings, consider saving these settings by method RsOsp.Configure.All.backup, before you send the restore command.

param restore_file_to_restore String parameter to select the backup file to be restored. The user interface ('WebGUI') shows only files that were saved in the default directory, hence, without specifying an additional file path.

Subgroups

5.1.6.1.1 Catalog

SCPI Commands

```
CONFigure:ALL:RESTore:CATalog
```

class Catalog

Catalog commands group definition. 1 total commands, 0 Sub-groups, 1 group commands

get(*restore_file_info: Optional[str] = None*) → List[str]

```
# SCPI: CONFigure:ALL:RESTore:CATalog
value: List[str] = driver.configure.all.restore.catalog.get(restore_file_info =
↪'1')
```

Queries the names of all backup files that are stored in the switch unit's internal flash memory. Each of these backup files comprises a full set of switch unit settings. All configuration filenames have the extension '.backup'. Do not enter the extension when specifying a filename. A filename query does not return the extension. For example, when you save the 'Interconnection' definition file 'settings-2018-10-25', it is saved as 'settings-2018-10-25.backup'. A query returns this filename as 'settings-2018-10-25', only.

param restore_file_info No help available

return list_of_backup_files: Comma-separated list of filenames, each in quotation marks. If no files exist, an empty string '' is returned.

5.1.7 Relay

class Relay

Relay commands group definition. 1 total commands, 1 Sub-groups, 0 group commands

Subgroups

5.1.7.1 Delay

SCPI Commands

```
CONFigure:RELay:DELay
```

class Delay

Delay commands group definition. 1 total commands, 0 Sub-groups, 1 group commands

get(*channel_list: str*) → List[int]

```
# SCPI: CONFigure:RELay:DELay
value: List[int] = driver.configure.relay.delay.get(channel_list = r1)
```

Sets or queries the command delay times for up to 4 external power-transfer relays connected to the R&S OSP-B104 Digital I/O Module (EMS) . The delay determines the period of time, which is reserved for a relay to change its state. Note that these external relays require switching times that are significantly longer than in most other relays. After receiving a method RsOsp.Route.Close.set command for changing the state of a connected external relay, the module R&S OSP-B104 behaves as follows:

INTRO_CMD_HELP: Saves all configuration settings to a backup file. These settings comprise the following:

- It sends the switching pulse during this full period of time
- Then it queries the relay's current position
- If the current position differs from the target position, the module generates a SCPI error, which is available via SYST:ERR?
- Then it accepts the next command

You can set delay times to ensure that the switching process of external transfer relays is completed, before further commands are executed. If you modify a delay time, the new value is stored durable on the module's EEPROM memory.

param channel_list List of external transfer relays (connected to module R&S OSP-B104) and associated delay times to be set or queried. - (@FxxMyy(sssee)): Defines the channel list for one relay with the following parameters:FxxMyy: as described in ROUTe:CLOSe.sss = 0 to 255. The digits in front of the last 2 digits (hence, the 1, 2 or 3 leading digits in the parenthesis) represent the 8-bit delay value. The adjustable delay time has a resolution of 50 ms and spans from 0 to 12.75 seconds (255 x 50 ms = 12750 ms) . The default value 2 is equivalent to a delay time of 100 ms.ee = 11, 12, 13, 14. The last 2 digits in the parenthesis represent the numbers ('names') of the relays (up to 4) that are connected to the R&S OSP-B104. The numeral offset of 10 distinguishes these relay numbers from the I/O channel numbers 01 to 04 on the same module.

return delay_factor_list: No help available

set(*channel_list: str*) → None

```
# SCPI: CONFigure:RELay:DELay
driver.configure.relay.delay.set(channel_list = r1)
```

Sets or queries the command delay times for up to 4 external power-transfer relays connected to the R&S OSP-B104 Digital I/O Module (EMS) . The delay determines the period of time, which is reserved for a relay to change its state. Note that these external relays require switching times that are significantly longer than in most other relays. After receiving a method RsOsp.Route.Close.set command for changing the state of a connected external relay, the module R&S OSP-B104 behaves as follows:

INTRO_CMD_HELP: Saves all configuration settings to a backup file. These settings comprise the following:

- It sends the switching pulse during this full period of time
- Then it queries the relay's current position
- If the current position differs from the target position, the module generates a SCPI error, which is available via SYST:ERR?
- Then it accepts the next command

You can set delay times to ensure that the switching process of external transfer relays is completed, before further commands are executed. If you modify a delay time, the new value is stored durable on the module's EEPROM memory.

param channel_list List of external transfer relays (connected to module R&S OSP-B104) and associated delay times to be set or queried. - (@FxxMyy(sssee)): Defines the channel list for one relay with the following parameters:FxxMyy: as described in ROUTe:CLOSe.sss = 0 to 255. The digits in front of the last 2 digits (hence, the 1, 2 or 3 leading digits in the parenthesis) represent the 8-bit delay value. The adjustable delay time has a resolution of 50 ms and spans from 0 to 12.75 seconds (255 x 50 ms = 12750 ms) . The default value 2 is equivalent to a delay time of 100 ms.ee = 11, 12, 13, 14. The last 2 digits in the parenthesis represent the numbers ('names') of the relays (up to 4) that are connected to the R&S OSP-B104. The numeral offset of 10 distinguishes these relay numbers from the I/O channel numbers 01 to 04 on the same module.

5.1.8 Lock

SCPI Commands

```
CONFigure:LOCK:MODE
```

class Lock

Lock commands group definition. 1 total commands, 0 Sub-groups, 1 group commands

get_mode() → bool

```
# SCPI: CONFigure:LOCK:MODE
value: bool = driver.configure.lock.get_mode()
```

Enables or disables the lock mode or queries this mode.

return state: - 1 | ON: The switching of relays and the setting of output channels is locked. - 0 | OFF: Relays and output channels are not locked.

set_mode(*state: bool*) → None

```
# SCPI: CONFigure:LOCK:MODE
driver.configure.lock.set_mode(state = False)
```

Enables or disables the lock mode or queries this mode.

param state

- 1 | ON: The switching of relays and the setting of output channels is locked.
- 0 | OFF: Relays and output channels are not locked.

5.2 Read

class Read

Read commands group definition. 3 total commands, 3 Sub-groups, 0 group commands

Subgroups

5.2.1 Io

class Io

Io commands group definition. 1 total commands, 1 Sub-groups, 0 group commands

Subgroups

5.2.1.1 InputPy

SCPI Commands

```
READ:IO:IN
```

class InputPy

InputPy commands group definition. 1 total commands, 0 Sub-groups, 1 group commands

get(*modules: str*) → List[int]

```
# SCPI: READ:IO:IN
value: List[int] = driver.read.io.inputPy.get(modules = r1)
```

Queries the states of all input channels of one or more selected modules. INTRO_CMD_HELP: The query applies only to modules that have I/O (input / output) channels. For example, the modules listed below have the following number of channels:

- R&S OSP-B103: 16 input channels
- R&S OSP-B104: 4 input channels

- R&S OSP-B114: 4 input channels

The return value is a set of single integer decimal numbers that represent the state of all queried input channels per module. Each integer is in the range of 0 to 65535:

INTRO_CMD_HELP: The query applies only to modules that have I/O (input / output) channels. For example, the modules listed below have the following number of channels:

- 0: all channels are logical 0 or low
- 65535: all channels (maximum 16) are logical 1 or high, where 65535 = 216-1

When converted to a binary number, each 1 or 0 digit shows the state of one channel. This representation starts with the lowest digit for channel one, up to the highest digit for channel 16. See also method **RsOsp.Read.Io.InputPy.get_**, method **RsOsp.Read.Io.InputPy.get_** and method **RsOsp.Read.Io.InputPy.get_**.

param modules Selects the modules that you want to query for their I/O channels. Identify the modules by their frame IDs Fxx and module numbers Myy. For a description of these parameters, refer to method RsOsp.Route.Close.set. Write the combined frame/module names FxxMyy, separated by commas, inside an expression of two brackets and the '@' sign. Do not use blank spaces or quotation marks in this expression. Example: (@F01M01,F01M06,F02M03) Only for querying one single module, you can use syntax without '(@...) ', for example: F01M01 If a module that you specify does not exist or does not support READ:IO:IN? (having no input channels) , the query returns no result and a SCPI error is generated. You can query the error with SYST:ERR?. For example, with a query READ:IO:IN? (@F01M06) , the result can be: -222,'Data out of range;Invalid index. frame F01: no module connected to M06,READ:IO:IN? F01M06' Or with a query READ:IO:IN? (@F01M03) , the result can be: -170,'Expression error;module on connector M03does not support input channels,READ:IO:IN? F01M03'

return input_channel_value: Comma-separated list of integer decimal numbers that represent the queried input channel states as described above.

get_multiple_modules(*modules: List[str]*) → List[int]
READ:IO:IN

Same as get_single_module(), but for multiple modules.

param modules Example value (without quotes): ['F01M03', 'F01M04']

get_single_module(*module: str*) → List[int]
READ:IO:IN

Same as get(), but you do not need to enter round brackets or the '@' character.

param module example value (without quotes): 'F01M03'

return input_channel_value: Comma-separated list of integer decimal numbers that represent the queried input channel states as described above.

5.2.2 Module

class Module

Module commands group definition. 1 total commands, 1 Sub-groups, 0 group commands

Subgroups

5.2.2.1 Interlock

SCPI Commands

```
READ:MODule:INTerlock
```

class Interlock

Interlock commands group definition. 1 total commands, 0 Sub-groups, 1 group commands

get(*modules: str*) → List[bool]

```
# SCPI: READ:MODule:INTerlock
value: List[bool] = driver.read.module.interlock.get(modules = r1)
```

Queries the interlock state of one or more selected modules. INTRO_CMD_HELP: The query applies only to modules that have an interlock, as in these modules:

- R&S OSP-B104
- R&S OSP-B114

param modules Selects the modules that you want to query for their interlock states. Identify the modules by their frame IDs Fxx and module numbers Myy. For a description of these parameters, refer to method RsOsp.Route.Close.set. Write the combined frame/module names FxxMyy, separated by commas, inside an expression of two brackets and the '@' sign. Do not use blank spaces or quotation marks in this expression. Example: (@F01M01,F01M06,F02M03) Only for querying one single module, you can use syntax without '(@...) ', for example: F01M01 If a module that you specify does not exist or does not support READ:MOD:INT? (having no interlock functionality) , the query returns no result and a SCPI error is generated. You can query the error with SYST:ERR?. For example, with a query READ:MOD:INT? (@F01M06) , the result can be: -222,'Data out of range;Invalid index. frame F01: no module connected to M06,READ:MOD:INT? F01M06' Or with a query READ:MOD:INT? (@F01M03) , the result can be: -170,'Expression error;module on connector M03does not support interlock,READ:MOD:INT? F01M03'

return interlock_state: Comma-separated list of '0 | 1' values that represent the interlock states. - 0: The interlock of the queried module is in open state, no measurements can be made. - 1: The interlock of the queried module is in closed state, measurements can proceed normally.

get_multiple_modules(*modules: List[str]*) → List[bool]

READ:MODule:INTerlock

Same as get_single_module(), but for multiple channels.

param modules Example value (without quotes): ['F01M03', 'F01M04']

get_single_module(*module: str*) → List[bool]
READ:MODule:INTerlock

Same as get(), but you do not need to enter round brackets or the '@' character.

param module example value (without quotes): 'F01M03'

5.2.3 Relay

class Relay
Relay commands group definition. 1 total commands, 1 Sub-groups, 0 group commands

Subgroups

5.2.3.1 Operations

SCPI Commands

```
READ:RELay:OPERations
```

class Operations
Operations commands group definition. 1 total commands, 0 Sub-groups, 1 group commands

get(*channel_list: str*) → List[int]

```
# SCPI: READ:RELay:OPERations
value: List[int] = driver.read.relay.operations.get(channel_list = r1)
```

Queries the internal switching counter, which acquires the total number of operation cycles of each relay (and even of I/O channels) . The number of cycles is stored durable in the flash EEPROM of the module that the relay is part of. Storing occurs after every hour of R&S OSP operation, but only if the number has changed. Besides this time-controlled storing, also the query command triggers storing the counter's value. To make sure not to lose any operation cycle counts, we recommend sending the command method **RsOsp.Read.Relay.Operations.get_** before terminating a remote control session. If the module that you specify does not have a switching counter, the query always returns the value '0' as the result.

INTRO_CMD_HELP: For example, the following solid-state relay (SSR) modules and digital I/O modules have no switching counter:

- R&S OSP-B103
- R&S OSP-B107
- R&S OSP-B127
- R&S OSP-B128
- R&S OSP-B142

In the R&S OSP-B104 and R&S OSP-B114, only the electromechanical relay has a switching counter.

param channel_list Specifies the relays and I/O channels to be read. For the channel list syntax, refer to method RsOsp.Route.Close.set.

return switch_counts: The query returns a comma-separated string with a number for each relay or channel in the list, in the same order as the channel list is specified.

get_multiple_channels(*channels: List[str]*) → List[int]
READ:RELay:OPERations

Same as get_single_channel(), but for multiple channels.

param channels Example value (without quotes): ['F01M03(0002)', 'F01M04(0003)']

get_single_channel(*channel: str*) → List[int]
READ:RELay:OPERations

Same as get(), but you do not need to enter round brackets or the '@' character.

param channel example value (without quotes): 'F01M03'

5.3 Route

class Route
Route commands group definition. 13 total commands, 4 Sub-groups, 0 group commands

Subgroups

5.3.1 Close

SCPI Commands

```
ROUTe:CLOSe
```

class Close
Close commands group definition. 1 total commands, 0 Sub-groups, 1 group commands

get(*channel_list_or_path_name: str*) → List[bool]

```
# SCPI: ROUTe:CLOSe
value: List[bool] = driver.route.close.get(channel_list_or_path_name = r1)
```

Sets or queries the state of selected relays or I/O channels. The query returns a 1 for each physical state of a queried relay or channel that is identical with the state specified in the channel list. If the physical state of a queried relay or channel differs from the state specified in the list, the query returns a 0. Note that for failsafe (monostable) relays, the query returns the state of the control line, only, while for latched (bistable) relays, the query always reads the true physical switching state. The parameter <channel list or path name> is also called the 'channel list string'. Its basic syntax is as follows: (@FxxMyy(sssee)) Defines the channel list for just one relay or I/O channel, with the following parameters:

INTRO_CMD_HELP: For example, the following solid-state relay (SSR) modules and digital I/O modules have no switching counter:

- xx = 01, 02, 03,. . . ,99 (frame ID in, e.g., switch unit name F01)
- yy = 01, 02, 03,. . . ,20 (module ID in, e.g., slot position M02) Note that the slot position is labeled with an M, although in the hardware the actual positions are either FS (front slot) or RS (rear slot) . Using M instead reflects the fact that the firmware can detect only to the motherboard connector, to which a module is connected. The actual front or rear mounting position is not detected. In a factory configuration, the correlation of slot positions and connectors follows the scheme in Figure 'Top view of the motherboard with its connectors for module

bus cables'. If you mount modules yourself, we recommend using the same correlation. Also note that the modules are addressed by the syntax M0x, as opposed to the syntax A1x that was used for the legacy switch units R&S OSP1x0. Setting commands accept both syntax versions, M0x or A1x. For query commands, to change from one to the other syntax version, use the command method RsOsp.Configure.Compatible.mode.

- sss = 000 … n (state of the element to be controlled in a module) The element can be a relay, an output channel or another Switchable item. Some system-specific or customer-specific modules can have different elements. In the string 'sss', you can omit leading zeros. Hence, '(@FxxMyy(ssee))' for double-digit states or '(@FxxMyy(see))' for single-digit states are permissible. The number of available states depends on the module type. Examples are 0 to 1 (for SPDT, DPDT and DP3T relays or I/O channels) , 0 to 6 (for SP6T and 4P6T) , or 0 to 8 (for SP8T) . Some modules, for example the 'R&S OSP-B104 Digital I/O Module (EMS) ', use sss to set a 3-digit state like the delay time. For details, refer to the description of the module.
- ee = 01 … m (number of the element to be controlled in a module) The number of available elements depends on the module type. Examples are 01 to 06 for the 6 SPDT relays in module R&S OSP-B101 or 01 to 16 for the 16 output channels in module R&S OSP-B103.

Some special modules also allow a different format, for example eee, if selecting the element requires 3 digits. For details, refer to the description of the module. If you want to address a series of relays or channels in the command method RsOsp.Route.Close.set, you can use one of the following concatenated syntax formats: (@FxxMyy(ssee) ,FxxMyy(ssee) , FxxMyy(ssee) ,…) Sets selected relays or channels in selected modules of selected switch units to the specified state. (In each element of the channel list, replace the parameters xx, yy, ss and ee with arbitrary numbers according to your needs.) Or concatenate addressing several relays or channels within a selected module: (@FxxMyy(sxex,syey,szez,…)) Sets several relays or channels (with numbers ex, ey, ez, …) in one module to individual states (sx, sy, sz, …) . For example, ROUT:CLOS (@F01M11(0102,0104,0105)) sets relays 2, 4 & 5 to state 1. (@FxxMyy(ssee:ssff)) Sets a continuous range of relays or channels in one module to the same state, with ff = ee + number of continuous relays. For example, ROUT:CLOS (@F01M11(0101:0105)) is equal to ROUT:CLOS (@F01M11(0101,0102,0103,0104,0105)).

param channel_list_or_path_name Channel list string as described above, specifying relays or channels and their states to be set or queried. Instead of an explicit channel list string, you can use a 'path name' (in quotation marks) , previously defined by method RsOsp.Route.Path.Define.set.

return arg_1: - 1: True, the relay or channel is in the state that is indicated in the channel list. - 0: False, the relay or channel is not in the state indicated in the channel list.

`get_multiple_channels`(*channels: List[str]*) → List[bool]
ROUTe:CLOSe

Same as get_single_channel(), but for multiple channels.

param channels Example value (without quotes): ['F01M01(0301)', 'F02M03(0101)']

`get_path`(*path_name: str*) → List[bool]
ROUTe:CLOSe

Instead of an explicit channel list string, you can use a "pathName" previously defined by the RsOsp.Route.Path.Define.set()

Parameters `path_name` – example of the path_name (without quotes): 'PathA'

Returns arg_1: - 1: True, the relay or channel is in the state that is indicated in the channel list. - 0: False, the relay or channel is not in the state indicated in the channel list.

get_single_channel(*channel: str*) → List[bool]
ROUTe:CLOSe

Same as get(), but you do not need to enter round brackets or the '@' character.

param channel example value (without quotes): 'F01M01(0301)'

set(*channel_list_or_path_name: str*) → None

```
# SCPI: ROUTe:CLOSe
driver.route.close.set(channel_list_or_path_name = r1)
```

Sets or queries the state of selected relays or I/O channels. The query returns a 1 for each physical state of a queried relay or channel that is identical with the state specified in the channel list. If the physical state of a queried relay or channel differs from the state specified in the list, the query returns a 0. Note that for failsafe (monostable) relays, the query returns the state of the control line, only, while for latched (bistable) relays, the query always reads the true physical switching state. The parameter <channel list or path name> is also called the 'channel list string'. Its basic syntax is as follows: (@FxxMyy(sssee)) Defines the channel list for just one relay or I/O channel, with the following parameters:

INTRO_CMD_HELP: For example, the following solid-state relay (SSR) modules and digital I/O modules have no switching counter:

- xx = 01, 02, 03,. . . ,99 (frame ID in, e.g., switch unit name F01)
- yy = 01, 02, 03,. . . ,20 (module ID in, e.g., slot position M02) Note that the slot position is labeled with an M, although in the hardware the actual positions are either FS (front slot) or RS (rear slot) . Using M instead reflects the fact that the firmware can detect only to the motherboard connector, to which a module is connected. The actual front or rear mounting position is not detected. In a factory configuration, the correlation of slot positions and connectors follows the scheme in Figure 'Top view of the motherboard with its connectors for module bus cables'. If you mount modules yourself, we recommend using the same correlation. Also note that the modules are addressed by the syntax M0x, as opposed to the syntax A1x that was used for the legacy switch units R&S OSP1x0. Setting commands accept both syntax versions, M0x or A1x. For query commands, to change from one to the other syntax version, use the command method RsOsp.Configure.Compatible.mode.
- sss = 000 . . . n (state of the element to be controlled in a module) The element can be a relay, an output channel or another Switchable item. Some system-specific or customer-specific modules can have different elements. In the string 'sss', you can omit leading zeros. Hence, '(@FxxMyy(ssee))' for double-digit states or '(@FxxMyy(see))' for single-digit states are permissible. The number of available states depends on the module type. Examples are 0 to 1 (for SPDT, DPDT and DP3T relays or I/O channels) , 0 to 6 (for SP6T and 4P6T) , or 0 to 8 (for SP8T) . Some modules, for example the 'R&S OSP-B104 Digital I/O Module (EMS) ', use sss to set a 3-digit state like the delay time. For details, refer to the description of the module.
- ee = 01 . . . m (number of the element to be controlled in a module) The number of available elements depends on the module type. Examples are 01 to 06 for the 6 SPDT relays in module R&S OSP-B101 or 01 to 16 for the 16 output channels in module R&S OSP-B103.

Some special modules also allow a different format, for example eee, if selecting the element requires 3 digits. For details, refer to the description of the module. If you want to address a series of relays or channels in the command method RsOsp.Route.Close.set, you can use one of the following concatenated syntax formats: (@FxxMyy(ssee) ,FxxMyy(ssee) , FxxMyy(ssee) ,. . .) Sets selected relays or channels in selected modules of selected switch units to the specified state. (In each element of the channel list, replace the parameters xx, yy, ss and ee with arbitrary numbers according to your needs.) Or concatenate addressing several relays or channels within a selected module: (@FxxMyy(sxex,syey,szez,. . .))

Sets several relays or channels (with numbers ex, ey, ez, . . .) in one module to individual states (sx, sy, sz, . . .) . For example, ROUT:CLOS (@F01M11(0102,0104,0105)) sets relays 2, 4 & 5 to state 1. (@FxxMyy(ssee:ssff)) Sets a continuous range of relays or channels in one module to the same state, with ff = ee + number of continuous relays. For example, ROUT:CLOS (@F01M11(0101:0105)) is equal to ROUT:CLOS (@F01M11(0101,0102,0103,0104,0105)).

param channel_list_or_path_name Channel list string as described above, specifying relays or channels and their states to be set or queried. Instead of an explicit channel list string, you can use a 'path name' (in quotation marks) , previously defined by method RsOsp.Route.Path.Define.set.

`set_multiple_channels`(*channels: List[str]*) → None
ROUTe:CLOSe

Same as set_single_channel(), but for multiple channels

Parameters **`channels`** – example value (without quotes): ['F01M01(0301)', 'F02M03(0101)']

`set_path`(*path_name: str*) → None
ROUTe:CLOSe

Instead of an explicit channel list string, you can use a "pathName" previously defined by the RsOsp.Route.Path.Define.set()

Parameters **`path_name`** – example of the path_name (without quotes): 'PathA'

`set_single_channel`(*channel: str*) → None
ROUTe:CLOSe

Same as set(), but you do not need to enter round brackets or the '@' character.

Parameters **`channel`** – example value (without quotes): 'F01M01(0301)'

5.3.2 Attenuation

SCPI Commands

```
ROUTe:ATTenuation
```

`class Attenuation`
Attenuation commands group definition. 1 total commands, 0 Sub-groups, 1 group commands

`get`(*channel_list: str*) → List[int]

```
# SCPI: ROUTe:ATTenuation
value: List[int] = driver.route.attenuation.get(channel_list = r1)
```

Sets or queries the level of attenuation, if a step attenuator is available in the module, for example in the R&S OSP-B171H and in the R&S OSP-B157WN. Also, you can control attenuators by the standard command method RsOsp.Route.Close. set, allowing joint control of attenuators and relays, and saving their target states in joint path definitions. Similar to method RsOsp.Route.Close.set, the parameter <channel list> uses the following syntax:

INTRO_CMD_HELP: (@FxxMyy(sssee))

- xx = 01, 02, 03,. . . ,99 (frame ID in, e.g., switch unit name F01)
- yy = 01, 02, 03,. . . ,20 (module ID in, e.g., slot position M02)

- sss = 000 … n (state of the attenuator to be controlled in a module)
- ee = 01 … m (element number of the attenuator to be controlled)

For example, in the R&S OSP-B171H: INTRO_CMD_HELP: (@FxxMyy(sssee))

- sss = 000 … 015 (attenuator steps)
- ee = 01 … 02 (attenuator number in module version .42) ee = 01 … 04 (attenuator number in module version .44)

Note that in the string 'sss', you can omit leading zeros.

param channel_list Channel list string as described above, selecting a module and attenuator and specifying the attenuation level to be set. The query also requires the element number 'ee', but it ignores the state information 'sss'. You can submit the value of 'sss' as 000 (or any arbitrary 3-digit value) , or you can omit it, entering only ee. The range and interpretation of the state value sss depends on the specific attenuator used in the module. For details, refer to the module description.

return attenuation_list: No help available

`get_multiple_channels`(*channels: List[str]*) → List[int]
ROUTe:ATTenuation

Same as get_single_channel(), but for multiple channels.

param channels Example value (without quotes): ['F01M03(0002)', 'F01M04(0003)']

`get_single_channel`(*channel: str*) → List[int]
ROUTe:ATTenuation

Same as get(), but you do not need to enter round brackets or the '@' character.

param channel example value (without quotes): 'F01M03(0001,0002,0003,0004)'

`set`(*channel_list: str*) → None

```
# SCPI: ROUTe:ATTenuation
driver.route.attenuation.set(channel_list = r1)
```

Sets or queries the level of attenuation, if a step attenuator is available in the module, for example in the R&S OSP-B171H and in the R&S OSP-B157WN. Also, you can control attenuators by the standard command method RsOsp.Route.Close. set, allowing joint control of attenuators and relays, and saving their target states in joint path definitions. Similar to method RsOsp.Route.Close.set, the parameter <channel list> uses the following syntax:

INTRO_CMD_HELP: (@FxxMyy(sssee))

- xx = 01, 02, 03,…,99 (frame ID in, e.g., switch unit name F01)
- yy = 01, 02, 03,…,20 (module ID in, e.g., slot position M02)
- sss = 000 … n (state of the attenuator to be controlled in a module)
- ee = 01 … m (element number of the attenuator to be controlled)

For example, in the R&S OSP-B171H: INTRO_CMD_HELP: (@FxxMyy(sssee))

- sss = 000 … 015 (attenuator steps)
- ee = 01 … 02 (attenuator number in module version .42) ee = 01 … 04 (attenuator number in module version .44)

Note that in the string 'sss', you can omit leading zeros.

param channel_list Channel list string as described above, selecting a module and attenuator and specifying the attenuation level to be set. The query also requires the element number 'ee', but it ignores the state information 'sss'. You can submit the value of 'sss' as 000 (or any arbitrary 3-digit value) , or you can omit it, entering only ee. The range and interpretation of the state value sss depends on the specific attenuator used in the module. For details, refer to the module description.

`set_multiple_channels`(*channels: List[str]*) → None
ROUTe:ATTenuation

Same as set_single_channel(), but for multiple channels

Parameters **`channels`** – example value (without quotes): ['F01M01(0301)', 'F02M03(0101)']

`set_single_channel`(*channel: str*) → None
ROUTe:ATTenuation

Same as set(), but you do not need to enter round brackets or the '@' character.

Parameters **`channel`** – example value (without quotes): 'F02M03(0101)'

5.3.3 Phase

SCPI Commands

```
ROUTe:PHASe
```

`class Phase`
Phase commands group definition. 1 total commands, 0 Sub-groups, 1 group commands

`get`(*channel_list: str*) → List[int]

```
# SCPI: ROUTe:PHASe
value: List[int] = driver.route.phase.get(channel_list = r1)
```

Sets or queries the phase angle, if a phase shifter is available in the module. Also, you can control phase shifters by the standard command method RsOsp.Route.Close.set, allowing joint control of phase shifters and relays, and saving their target states in joint path definitions. Similar to method RsOsp.Route.Close.set, the parameter <channel list> uses the following syntax:

INTRO_CMD_HELP: (@FxxMyy(sssee))

- xx = 01, 02, 03,…,99 (frame ID in, e.g., switch unit name F01)
- yy = 01, 02, 03,…,20 (module ID in, e.g., slot position M02)
- sss = 000 … n (state of the phase shifter to be controlled in a module)
- ee = 01 … m (element number of the phase shifter to be controlled)

Note that in the string 'sss', you can omit leading zeros.

param channel_list Channel list string as described above, selecting a module and phase shifter and specifying the phase angle to be set. The query also requires the element number 'ee', but it ignores the state information 'sss'. You can submit the value of 'sss' as 000 (or any arbitrary 3-digit value) , or you can omit it, entering only ee. The range and interpretation of the state value sss depends on the specific phase shifter used in the module. For details, refer to the module description.

return phase_list: No help available

get_multiple_channels(*channels: List[str]*) → List[int]
ROUTe:PHASe

Same as get_single_channel(), but for multiple channels.

param channels Example value (without quotes): [‘F01M03(0002)’, ‘F01M04(0003)’]

get_single_channel(*channel: str*) → List[int]
ROUTe:PHASe

Same as get(), but you do not need to enter round brackets or the ‘@’ character.

param channel example value (without quotes): ‘F01M03(0001,0002,0003,0004)’

set(*channel_list: str*) → None

```
# SCPI: ROUTe:PHASe
driver.route.phase.set(channel_list = r1)
```

Sets or queries the phase angle, if a phase shifter is available in the module. Also, you can control phase shifters by the standard command method RsOsp.Route.Close.set, allowing joint control of phase shifters and relays, and saving their target states in joint path definitions. Similar to method RsOsp.Route.Close.set, the parameter <channel list> uses the following syntax:

INTRO_CMD_HELP: (@FxxMyy(sssee))

- xx = 01, 02, 03,. . . ,99 (frame ID in, e.g., switch unit name F01)
- yy = 01, 02, 03,. . . ,20 (module ID in, e.g., slot position M02)
- sss = 000 . . . n (state of the phase shifter to be controlled in a module)
- ee = 01 . . . m (element number of the phase shifter to be controlled)

Note that in the string ‘sss’, you can omit leading zeros.

param channel_list Channel list string as described above, selecting a module and phase shifter and specifying the phase angle to be set. The query also requires the element number ‘ee’, but it ignores the state information ‘sss’. You can submit the value of ‘sss’ as 000 (or any arbitrary 3-digit value) , or you can omit it, entering only ee. The range and interpretation of the state value sss depends on the specific phase shifter used in the module. For details, refer to the module description.

5.3.4 Path

SCPI Commands

```
ROUTe:PATH:CATalog
ROUTe:PATH:LAST
ROUTe:PATH:DELete:ALL
ROUTe:PATH:EXPort
```

class Path
Path commands group definition. 10 total commands, 3 Sub-groups, 4 group commands

delete_all() → None

```
# SCPI: ROUTe:PATH:DELete:ALL
driver.route.path.delete_all()
```

Deletes all previously defined paths from the switch unit's internal volatile memory. Before you delete all paths, we recommend using the command method RsOsp.Route.Path.catalog to query all currently defined path names.

delete_all_with_opc() → None

```
# SCPI: ROUTe:PATH:DELete:ALL
driver.route.path.delete_all_with_opc()
```

Deletes all previously defined paths from the switch unit's internal volatile memory. Before you delete all paths, we recommend using the command method RsOsp.Route.Path.catalog to query all currently defined path names.

Same as delete_all, but waits for the operation to complete before continuing further. Use the RsOsp.utilities.opc_timeout_set() to set the timeout value.

export(*path_config_file: str*) → None

```
# SCPI: ROUTe:PATH:EXPort
driver.route.path.export(path_config_file = '1')
```

Stores a nonvolatile file in the compact flash memory of an R&S OSP by transferring it from the instrument's internal volatile memory. The stored file comprises all currently defined path configurations, see method RsOsp.Route.Path.Define. set. All path filenames have the extension '.path'. Do not enter the extension when specifying a filename. A filename query does not return the extension. For example, when you save the path file 'gen-pa_1', it is saved as 'gen-pa_1.path'. A query returns this filename as 'gen-pa_1', only. The command MMEM:STORe:STATe is equivalent with method RsOsp.Route. Path.export.

param path_config_file String parameter to specify the name of the file to be stored.

get_catalog() → List[str]

```
# SCPI: ROUTe:PATH:CATalog
value: List[str] = driver.route.path.get_catalog()
```

Returns a list of all currently defined path names in the internal volatile memory of the switch unit. The query addresses the default directory path '/home/instrument/ospdata' in the internal storage of the R&S OSP, unless you specify a different directory path, which is optional.

return list_of_path_names: Comma-separated list of path names, each in quotation marks.

get_last() → str

```
# SCPI: ROUTe:PATH:LAST
value: str = driver.route.path.get_last()
```

Queries the name of the previously switched path. If the previous switching action was based on method RsOsp.Route.Close. set + channel list string (rather than a path name) , the response is <Individual Settings>. After a *RST command, the response is <Reset State>. In the main page of the switch unit's user interface, the line Last Switched Path (enabled by the 'Path Info' setting or by method RsOsp.Configure.MainInfo.path) shows the same information as the response to method RsOsp.Route.Path.last.

return path_name: See method RsOsp.Route.Path.Define.set

Subgroups

5.3.4.1 Define

SCPI Commands

```
ROUTe:PATH:DEFine
```

class Define

Define commands group definition. 1 total commands, 0 Sub-groups, 1 group commands

List

alias of List

get(*path_name: str*) → str

```
# SCPI: ROUTe:PATH[:DEFine]
value: str = driver.route.path.define.get(path_name = '1')
```

method RsOsp.Route.Path.Define.set or method RsOsp.Route.Path.Define.set defines a path name and the channel-list string that can be replaced by this path name. A short path name can thus represent a long list of specific states of relays and I/O channels. Use method RsOsp.Route.Close.set to switch a path. The query returns the channel list that encodes the defined states for all relays and I/O channels in this path. Note that in 'Compatibility Mode', the query returns a string with syntax that differs from the channel list (see query example below) .

param path_name String parameter to specify the name of the path to be defined or queried. Limited to a maximum of 35 characters. Write the path name in quotation marks. The firmware observes capitalization of the path name. For example, 'path a' in lower case is not the same as 'Path A' in upper and lower case. A newly defined path name only exists in the instruments internal volatile memory (RAM) . At shutdown, all path definitions are saved permanently in the instrument's flash memory At startup, all saved path definitions are restored automatically. All new path definitions, which you made since the last startup, are lost, if you switch off the device by the rear on/off switch. The same holds true, if you switch off the device by pushing the front PWR key for more than 10 seconds, or if the firmware crashes. You can trigger immediate storing of all defined path names in the instrument's flash memory by using the command method RsOsp.Route.Path.export.

return channel_list: List of relays and I/O channels and their states to be set, as described in method RsOsp.Route.Close.set.

set(*path_name: str*, *channel_list: str*) → None

```
# SCPI: ROUTe:PATH[:DEFine]
driver.route.path.define.set(path_name = '1', channel_list = r1)
```

method RsOsp.Route.Path.Define.set or method RsOsp.Route.Path.Define.set defines a path name and the channel-list string that can be replaced by this path name. A short path name can thus represent a long list of specific states of relays and I/O channels. Use method RsOsp.Route.Close.set to switch a path. The query returns the channel list that encodes the defined states for all relays and I/O channels in this path. Note that in 'Compatibility Mode', the query returns a string with syntax that differs from the channel list (see query example below) .

param path_name String parameter to specify the name of the path to be defined or queried. Limited to a maximum of 35 characters. Write the path name in quotation marks. The firmware observes capitalization of the path name. For example, 'path a' in lower case is not the same as 'Path A' in upper and lower case. A newly defined path name only exists in the instruments internal volatile memory (RAM) . At shutdown, all path definitions are saved permanently in the instrument's flash memory At startup, all saved path definitions are restored automatically. All new path definitions, which you made since the last startup, are lost, if you switch off the device by the rear on/off switch. The same holds true, if you switch off the device by pushing the front PWR key for more than 10 seconds, or if the firmware crashes. You can trigger immediate storing of all defined path names in the instrument's flash memory by using the command method RsOsp.Route.Path.export.

param channel_list List of relays and I/O channels and their states to be set, as described in method RsOsp.Route.Close.set.

`set_multiple_channels`(*path_name: str*, *channels: List[str]*) → None

ROUTe:PATH[:DEFine]

Same as set_single_channel(), but for multiple channels

Parameters

- `path_name` – String parameter to specify the name of the path to be defined
- `channels` – example value (without quotes): ['F01M01(0301)', 'F02M03(0101)']

`set_single_channel`(*path_name: str*, *channel: str*) → None

ROUTe:PATH[:DEFine]

Same as set(), but you do not need to enter round brackets or the '@' character.

Parameters

- `path_name` – String parameter to specify the name of the path to be defined
- `channel` – example value (without quotes): 'F02M03(0101)'

5.3.4.2 Delete

SCPI Commands

```
ROUTe:PATH:DELete:NAME
```

`class Delete`

Delete commands group definition. 1 total commands, 0 Sub-groups, 1 group commands

`set_name`(*path_name: str*) → None

```
# SCPI: ROUTe:PATH:DELete[:NAME]
driver.route.path.delete.set_name(path_name = '1')
```

Deletes the path specified by the <path name> parameter from the switch unit's internal volatile memory. If this path does not exist, the command has no effect.

param path_name See method RsOsp.Route.Path.Define.set.

5.3.4.3 ImportPy

SCPI Commands

```
ROUTe:PATH:IMPort:DELete
ROUTe:PATH:IMPort:DELete:ALL
ROUTe:PATH:IMPort
```

class ImportPy

ImportPy commands group definition. 4 total commands, 1 Sub-groups, 3 group commands

class ValueStruct

Structure for setting input parameters. Contains optional set arguments. Fields:

- Import_Filename: str: String parameter to specify the name of the file to be loaded.
- **Replace_Or_Keep: enums.ReplaceOrKeep: Optional setting parameter. Optional parameter that decides about**
  - KEEP: Amends the current path definitions in the switch unit's internal memory with the imported path definitions.However, if you import paths that have the same names as existing paths in the memory, the imported paths overwrite the existing paths, even if you have specified to KEEP them.
  - REPLace: Discards the current path definitions in the switch unit's internal memory and replaces them with the imported path definitions.

delete(*path_config_file: str*) → None

```
# SCPI: ROUTe:PATH:IMPort:DELete
driver.route.path.importPy.delete(path_config_file = '1')
```

Risk of losing settings: Removes the specified path configuration file from the switch unit's compact flash memory. All path filenames have the extension '.path'. Do not enter the extension when specifying a filename. A filename query does not return the extension. For example, when you save the path file 'gen-pa_1', it is saved as 'gen-pa_1.path'. A query returns this filename as 'gen-pa_1', only. If the specified file does not exist, a SCPI error is generated. You can query the error with SYST:ERR?. The result can be, for example: -200,'Execution error;File does not exist. ,ROUTe:PATH:IMPort:DELete ''Path5''' The command MMEMory:DELete is equivalent with method RsOsp.Route.Path.ImportPy. delete.

param path_config_file String parameter to specify the name of the file to be deleted.

delete_all(*path_information: Optional[str] = None*) → None

```
# SCPI: ROUTe:PATH:IMPort:DELete:ALL
driver.route.path.importPy.delete_all(path_information = '1')
```

Risk of losing settings: Removes all path configuration files from the switch unit's compact flash memory. Before you delete all path configuration files, we recommend using the command method RsOsp.Route.Path.ImportPy.Catalog. get_ to query all currently defined path configuration files.

param path_information No help available

set_value(*value: RsOsp.Implementations.Route_.Path_.ImportPy.ImportPy.ValueStruct*) → None

```
# SCPI: ROUTe:PATH:IMPort
driver.route.path.importPy.set_value(value = ValueStruct())
```

Loads a set of path configurations from a file on the compact flash memory into the switch unit's internal volatile memory. All path filenames have the extension '.path'. Do not enter the extension when specifying a filename. A filename query does not return the extension. For example, when you save the path file 'gen-pa_1', it is saved as 'gen-pa_1.path'. A query returns this filename as 'gen-pa_1', only. If the specified file does not exist, a SCPI error is generated. You can query the error with SYST:ERR?. The result can be, for example: -200,'Execution error;Restoring device from file /opt/ospn/exportPath5.path failed,ROUTe:PATH:IMPort ''Path5''' The legacy command MMEM:LOAD:STATe is equivalent with method RsOsp.Route.Path.ImportPy.value. However, MMEM:LOAD:STATe does not support the parameter <import mode>, which is used with method RsOsp.Route.Path.ImportPy.value to specify keeping or replacing the path definitions (see below) . Risk of losing settings: Note that this command overwrites all current path definitions in the switch unit's internal volatile memory with the path definitions in the loaded file. To avoid losing current path definitions, consider saving these definitions by method RsOsp.Route.Path.export, before you send the import command.

param value see the help for ValueStruct structure arguments.

Subgroups

5.3.4.3.1 Catalog

SCPI Commands

```
ROUTe:PATH:IMPort:CATalog
```

class Catalog

Catalog commands group definition. 1 total commands, 0 Sub-groups, 1 group commands

get(*path_information: Optional[str] = None*) → List[str]

```
# SCPI: ROUTe:PATH:IMPort:CATalog
value: List[str] = driver.route.path.importPy.catalog.get(path_information = '1
↪')
```

Returns the names of all switching path configuration files that are stored in the switch unit's flash memory. All path filenames have the extension '.path'. Do not enter the extension when specifying a filename. A filename query does not return the extension. For example, when you save the path file 'gen-pa_1', it is saved as 'gen-pa_1.path'. A query returns this filename as 'gen-pa_1', only. The command MMEMory:CATalog? is equivalent with method RsOsp.Route. Path.ImportPy.Catalog.get_

param path_information No help available

return list_of_exp_path_configs: Comma-separated list of filenames, each in quotation marks.

5.4 Diagnostic

class Diagnostic

Diagnostic commands group definition. 6 total commands, 1 Sub-groups, 0 group commands

Subgroups

5.4.1 Service

SCPI Commands

```
DIAGnostic:SERVice:HWINfo
```

class Service

Service commands group definition. 6 total commands, 2 Sub-groups, 1 group commands

get_hw_info() → List[str]

```
# SCPI: DIAGnostic:SERVice:HWINfo
value: List[str] = driver.diagnostic.service.get_hw_info()
```

Returns information about all hardware components (motherboards and modules) that are part of the complete system of one or several R&S OSP instruments.

INTRO_CMD_HELP: The returned component information consists of:

- Location ID (= switch unit and module number, for example, frame F01 is the primary device, frame F02 is the first secondary device, M00 is the primary device's motherboard, M01 is the module connected to the connector M01)
- Name (for example, OSPMAINBOARD, OSP220, OSP230, OSP320, OSP-B101)
- Serial number (for example, 100173)
- Part number (= order number, for example, 1528.3105.03)
- Hardware code:

INTRO_CMD_HELP: The returned component information consists of:

- Modules that are controlled via 1 module bus typically return the code 0
- Modules that are controlled via 2 module buses return the codes 1 for the first control board and 2 for the second control board
- Product index (model iteration of a hardware version, for example, 01.00)

return hw_info_list: The response is a string in following format: location|name|sn_nbr|part_nbr| hardware_code|product_index

Subgroups

5.4.1.1 Module

class Module

Module commands group definition. 3 total commands, 3 Sub-groups, 0 group commands

Subgroups

5.4.1.1.1 HwInfo

SCPI Commands

```
DIAGnostic:SERVice:MODule:HWINfo
```

class HwInfo

HwInfo commands group definition. 1 total commands, 0 Sub-groups, 1 group commands

get(*modules: str*) → List[str]

```
# SCPI: DIAGnostic:SERVice:MODule:HWINfo
value: List[str] = driver.diagnostic.service.module.hwInfo.get(modules = r1)
```

The setting command can make modules update their hardware configuration. The command is implemented for all modules, but it was developed especially for system modules that can have submodules attached. The command allows updating such submodules in the firmware during operation. For modules that cannot update any hardware configuration, the command has no effect. The query command returns the most recently updated hardware configuration. Modules, which support this query, reply with an individual, module-specific return string. All other modules reply with an empty string ‘’.

param modules The command addresses the modules by the following syntax, like the module names string in method **RsOsp.Read.Io.InputPy.get_**: xx = 01, 02, 03,. . . ,99 (frame ID in, e.g., switch unit name F01) yy = 01, 02, 03,. . . ,20 (module ID in, e.g., slot position M02)

return hw_info: No help available

set(*modules: str*) → None

```
# SCPI: DIAGnostic:SERVice:MODule:HWINfo
driver.diagnostic.service.module.hwInfo.set(modules = r1)
```

The setting command can make modules update their hardware configuration. The command is implemented for all modules, but it was developed especially for system modules that can have submodules attached. The command allows updating such submodules in the firmware during operation. For modules that cannot update any hardware configuration, the command has no effect. The query command returns the most recently updated hardware configuration. Modules, which support this query, reply with an individual, module-specific return string. All other modules reply with an empty string ‘’.

param modules The command addresses the modules by the following syntax, like the module names string in method **RsOsp.Read.Io.InputPy.get_**: xx = 01, 02, 03,. . . ,99 (frame ID in, e.g., switch unit name F01) yy = 01, 02, 03,. . . ,20 (module ID in, e.g., slot position M02)

5.4.1.1.2 Temperature

SCPI Commands

```
DIAGnostic:SERVice:MODule:TEMPerature
```

class Temperature

Temperature commands group definition. 1 total commands, 0 Sub-groups, 1 group commands

get(*modules: str*) → List[float]

```
# SCPI: DIAGnostic:SERVice:MODule:TEMPerature
value: List[float] = driver.diagnostic.service.module.temperature.get(modules =␣
↪r1)
```

Queries the temperature of selected modules in degrees centigrade.

param modules The command addresses the modules by the following syntax, like the module names string in method **RsOsp.Read.Io.InputPy.get_**: xx = 01, 02, 03,. . . ,99 (frame ID in, e.g., switch unit name F01) yy = 01, 02, 03,. . . ,20 (module ID in, e.g., slot position M02)

return temperature_value: No help available

5.4.1.1.3 Function

SCPI Commands

```
DIAGnostic:SERVice:MODule:FUNCtion
```

class Function

Function commands group definition. 1 total commands, 0 Sub-groups, 1 group commands

get(*modules: str*, *function: str*) → List[str]

```
# SCPI: DIAGnostic:SERVice:MODule:FUNCtion
value: List[str] = driver.diagnostic.service.module.function.get(modules = r1,␣
↪function = '1')
```

No command help available

param modules No help available

param function No help available

return response: No help available

set(*modules: str*, *function: str*) → None

```
# SCPI: DIAGnostic:SERVice:MODule:FUNCtion
driver.diagnostic.service.module.function.set(modules = r1, function = '1')
```

No command help available

param modules No help available

param function No help available

5.4.1.2 User

SCPI Commands

```
DIAGnostic:SERVice:USER:ERRor
DIAGnostic:SERVice:USER:WARNing
```

class User

User commands group definition. 2 total commands, 0 Sub-groups, 2 group commands

get_error() → List[str]

```
# SCPI: DIAGnostic:SERVice:USER:ERRor
value: List[str] = driver.diagnostic.service.user.get_error()
```

Queries for device errors that were not necessarily evoked by a remote control command. Typically, use this query for errors that may have come up during booting.

return user_error_list: No help available

get_warning_py() → List[str]

```
# SCPI: DIAGnostic:SERVice:USER:WARNing
value: List[str] = driver.diagnostic.service.user.get_warning_py()
```

Queries for device warnings that were not necessarily evoked by a remote control command. Typically, use this query for warnings that may have come up during booting.

return user_warning_list: No help available

5.5 Trigger

SCPI Commands

```
TRIGger:STATe
TRIGger:TYPE
TRIGger:INDex
TRIGger:EXPort
```

class Trigger

Trigger commands group definition. 17 total commands, 5 Sub-groups, 4 group commands

export(*trigger_config_file: str*) → None

```
# SCPI: TRIGger:EXPort
driver.trigger.export(trigger_config_file = '1')
```

Stores the currently defined trigger configuration as a nonvolatile file in the compact flash memory of your primary switch unit.

param trigger_config_file String parameter to specify the name of the file to be stored.

get_index() → int

```
# SCPI: TRIGger:INDex
value: int = driver.trigger.get_index()
```

Queries the trigger index, which is the number of the currently triggered path in the trigger types described below.

INTRO_CMD_HELP: The returned component information consists of:

- 'Toggle A-B', the index has the following meaning:

INTRO_CMD_HELP: The returned component information consists of:

- -1 = no trigger event yet, method RsOsp.Trigger.Count.value = 0
- 0 = Path A
- 1 = Path B
- 'Sequenced', the index has the following meaning:

INTRO_CMD_HELP: The returned component information consists of:

- -1 = no trigger event yet, method RsOsp.Trigger.Count.value = 0
- 0 = Path 0
- 1 = Path 1
- 2 = Path 2
- …
- 15 = Path 15

return index: No help available

get_state() → bool

```
# SCPI: TRIGger:STATe
value: bool = driver.trigger.get_state()
```

Sets or queries the activation state of the trigger functionality.

return activation_state: No help available

get_type_py() → RsOsp.enums.TriggerType

```
# SCPI: TRIGger:TYPE
value: enums.TriggerType = driver.trigger.get_type_py()
```

Selects or queries the trigger type.

return type_py: No help available

set_state(*activation_state: bool*) → None

```
# SCPI: TRIGger:STATe
driver.trigger.set_state(activation_state = False)
```

Sets or queries the activation state of the trigger functionality.

param activation_state

- OFF: Deactivates the trigger functionality. The command does not accept '0' instead of 'OFF'.
- ON: Activates the trigger functionality. The command does not accept '1' instead of 'ON'.

set_type_py(*type_py: RsOsp.enums.TriggerType*) → None

```
# SCPI: TRIGger:TYPE
driver.trigger.set_type_py(type_py = enums.TriggerType.ADDRessed)
```

Selects or queries the trigger type.

param type_py

- SINGle: Selects the trigger type Single.
- TOGGle: Selects the trigger type Toggle A-B.
- SEQuenced: Selects the trigger type Sequenced.
- ADDRessed: Selects the trigger type Addressed.

Subgroups

5.5.1 Signal

SCPI Commands

```
TRIGger:SIGNal:LEVel
TRIGger:SIGNal:SLOPe
TRIGger:SIGNal:TERMination
```

class Signal

Signal commands group definition. 3 total commands, 0 Sub-groups, 3 group commands

get_level() → float

```
# SCPI: TRIGger:SIGNal:LEVel
value: float = driver.trigger.signal.get_level()
```

Sets or queries the voltage level, which defines the threshold for a trigger event. The level setting is not influenced by reset commands.

return level: No help available

get_slope() → RsOsp.enums.TriggerSlope

```
# SCPI: TRIGger:SIGNal:SLOPe
value: enums.TriggerSlope = driver.trigger.signal.get_slope()
```

Sets or queries, which kind of threshold transition is interpreted as a trigger event. For the threshold level, see method RsOsp.Trigger.Signal.level.

return slope: - POSitive: Defines interpreting a positive (low-to-high) transition of the threshold as a trigger event. - NEGative: Defines interpreting a negative (high-to-low) transition of the threshold as a trigger event. - BOTH: Defines interpreting a positive or a negative transition of the threshold as a trigger event.Equivalent with Any in the user interface ('WebGUI') .

get_termination() → float

```
# SCPI: TRIGger:SIGNal:TERMination
value: float = driver.trigger.signal.get_termination()
```

Sets or queries the type of termination of the trigger signal cable.

return termination: No help available

set_level(*level: float*) → None

```
# SCPI: TRIGger:SIGNal:LEVel
driver.trigger.signal.set_level(level = 1.0)
```

Sets or queries the voltage level, which defines the threshold for a trigger event. The level setting is not influenced by reset commands.

param level Specifies the voltage level. Factory default is 2.5 V.

set_slope(*slope: RsOsp.enums.TriggerSlope*) → None

```
# SCPI: TRIGger:SIGNal:SLOPe
driver.trigger.signal.set_slope(slope = enums.TriggerSlope.BOTH)
```

Sets or queries, which kind of threshold transition is interpreted as a trigger event. For the threshold level, see method RsOsp.Trigger.Signal.level.

param slope

- POSitive: Defines interpreting a positive (low-to-high) transition of the threshold as a trigger event.
- NEGative: Defines interpreting a negative (high-to-low) transition of the threshold as a trigger event.
- BOTH: Defines interpreting a positive or a negative transition of the threshold as a trigger event.Equivalent with Any in the user interface ('WebGUI') .

set_termination(*termination: float*) → None

```
# SCPI: TRIGger:SIGNal:TERMination
driver.trigger.signal.set_termination(termination = 1.0)
```

Sets or queries the type of termination of the trigger signal cable.

param termination

- 50OHM: Sets the termination to 50 Ohm.Also accepted parameter syntax: '50' or '50 ohm' (a blank and all capital/small letters are ignored) .
- HIGH: Sets the termination to High, equivalent with High Impedance in the user interface ('WebGUI') .Also accepted parameter syntax: 'high' (capital/small letters are ignored) .

5.5.2 Sequence

class Sequence

Sequence commands group definition. 3 total commands, 1 Sub-groups, 0 group commands

Subgroups

5.5.2.1 Define

SCPI Commands

```
TRIGger:SEQuence:DEFine:ALL
TRIGger:SEQuence:DEFine:LENGth
```

class Define

Define commands group definition. 3 total commands, 1 Sub-groups, 2 group commands

get_all() → List[str]

```
# SCPI: TRIGger:SEQuence:DEFine:ALL
value: List[str] = driver.trigger.sequence.define.get_all()
```

Sets several or all paths for the sequenced trigger. The query returns all path names.

return path_name: No help available

get_length() → int

```
# SCPI: TRIGger:SEQuence:DEFine:LENGth
value: int = driver.trigger.sequence.define.get_length()
```

Sets or queries the length of the sequence for the Sequenced trigger.

return length: No help available

set_all(*path_name: List[str]*) → None

```
# SCPI: TRIGger:SEQuence:DEFine:ALL
driver.trigger.sequence.define.set_all(path_name = ['1', '2', '3'])
```

Sets several or all paths for the sequenced trigger. The query returns all path names.

param path_name Specifies all path names for the trigger sequence. Separate the path names by commas and write each path name in quotation marks.

set_length(*length: int*) → None

```
# SCPI: TRIGger:SEQuence:DEFine:LENGth
driver.trigger.sequence.define.set_length(length = 1)
```

Sets or queries the length of the sequence for the Sequenced trigger.

param length No help available

Subgroups

5.5.2.1.1 Entry

SCPI Commands

```
TRIGger:SEQuence:DEFine:ENTRy
```

class Entry

Entry commands group definition. 1 total commands, 0 Sub-groups, 1 group commands

get(*index: int*) → str

```
# SCPI: TRIGger:SEQuence:DEFine[:ENTRy]
value: str = driver.trigger.sequence.define.entry.get(index = 1)
```

Sets or queries the path, which is defined for the selected number in the sequence.

param index Selects a number in the trigger sequence.

return path_name: No help available

set(*index: int*, *path_name: str*) → None

```
# SCPI: TRIGger:SEQuence:DEFine[:ENTRy]
driver.trigger.sequence.define.entry.set(index = 1, path_name = '1')
```

Sets or queries the path, which is defined for the selected number in the sequence.

param index Selects a number in the trigger sequence.

param path_name Specifies the path name to be set for the selected number in the trigger sequence. Write the path name in quotation marks.

5.5.3 Execute

SCPI Commands

```
TRIGger:EXECute
```

class Execute

Execute commands group definition. 1 total commands, 0 Sub-groups, 1 group commands

set(*type_py: Optional[RsOsp.enums.TriggerExecType] = None*) → None

```
# SCPI: TRIGger:EXECute
driver.trigger.execute.set(type_py = enums.TriggerExecType.RESet)
```

Sends a software trigger event or resets the trigger sequence.

param type_py

- TRIGger: TRIG:EXEC TRIG sends software trigger event, equivalent with the 'Manual Trigger'.
- RESet: TRIG:EXEC RES resets the sequence of the 'Sequenced' trigger.

5.5.4 Count

SCPI Commands

```
TRIGger:COUNt:VALue
TRIGger:COUNt:OVERflow
```

class Count

Count commands group definition. 2 total commands, 0 Sub-groups, 2 group commands

get_overflow() → bool

```
# SCPI: TRIGger:COUNt:OVERflow
value: bool = driver.trigger.count.get_overflow()
```

Queries, if a trigger overflow has happened since the last activation of the trigger functionality. If the trigger input connectors of the switch unit receive input signals faster than the firmware can process, it cannot count all trigger events, and it cannot update the trigger counter correctly. You can use the command to check, if this case has occurred since the last trigger activation.

return overflow: No help available

get_value() → int

```
# SCPI: TRIGger:COUNt[:VALue]
value: int = driver.trigger.count.get_value()
```

Queries the trigger count, hence the number of executed trigger events since the last activation of the trigger functionality.

return count: No help available

5.5.5 ImportPy

SCPI Commands

```
TRIGger:IMPort:DELete
TRIGger:IMPort:DELete:ALL
TRIGger:IMPort
```

class ImportPy

ImportPy commands group definition. 4 total commands, 1 Sub-groups, 3 group commands

delete(*trigger_config_file: str*) → None

```
# SCPI: TRIGger:IMPort:DELete
driver.trigger.importPy.delete(trigger_config_file = '1')
```

Risk of losing settings: Removes the specified trigger configuration file from the primary switch unit's compact flash memory. All trigger configuration filenames have the extension '.trigger'. Do not enter the extension when specifying a filename. A filename query does not return the extension. For example, when you save the trigger configuration file 'trg42', it is saved as 'trg42.trigger'. A query returns this filename as 'trg42', only.

param trigger_config_file String parameter to specify the name of the file to be deleted. If this file does not exist, a SCPI error is generated. You can query the error with SYST:ERR?. The result can be, for example: -200,'Execution error;File does not exist.,TRIG:IMP:DEL ''sequencedtrig2''

delete_all(*path_information: Optional[str] = None*) → None

```
# SCPI: TRIGger:IMPort:DELete:ALL
driver.trigger.importPy.delete_all(path_information = '1')
```

Risk of losing settings: Removes all trigger configuration files from the primary switch unit's compact flash memory. Before you delete all trigger configuration files, we recommend using the command method RsOsp.Trigger.ImportPy.Catalog. get_ to query all currently stored trigger configuration files. All trigger configuration filenames have the extension '. trigger'. Do not enter the extension when specifying a filename. A filename query does not return the extension. For example, when you save the trigger configuration file 'trg42', it is saved as 'trg42.trigger'. A query returns this filename as 'trg42', only.

param path_information No help available

set_value(*import_filename: str*) → None

```
# SCPI: TRIGger:IMPort
driver.trigger.importPy.set_value(import_filename = '1')
```

Loads a trigger configuration file from the compact flash memory of your primary switch unit into its internal volatile memory. As a prerequisite, you must have exported such a file in advance, see method RsOsp.Trigger.export. All trigger configuration filenames have the extension '.trigger'. Do not enter the extension when specifying a filename. A filename query does not return the extension. For example, when you save the trigger configuration file 'trg42', it is saved as 'trg42.trigger'. A query returns this filename as 'trg42', only. Risk of losing settings: Note that this command overwrites the current trigger configuration in the primary switch unit's internal memory with the trigger configuration in the loaded file. To avoid losing a current trigger configuration, consider saving this configuration by method RsOsp. Trigger.export, before you send the import command.

param import_filename String parameter to specify the name of the file to be stored.

Subgroups

5.5.5.1 Catalog

SCPI Commands

```
TRIGger:IMPort:CATalog
```

class Catalog

Catalog commands group definition. 1 total commands, 0 Sub-groups, 1 group commands

get(*path_information: Optional[str] = None*) → List[str]

```
# SCPI: TRIGger:IMPort:CATalog
value: List[str] = driver.trigger.importPy.catalog.get(path_information = '1')
```

Returns a list of trigger configurations that are stored in the primary switch unit's volatile memory.

param path_information The reply is a comma-separated list of the exported trigger configuration filenames.

return list_of_exp_trigger_configs: No help available

RSOSP UTILITIES

class Utilities

Common utility class. Utility functions common for all types of drivers.

Access snippet: `utils = RsOsp.utilities`

property logger: RsOsp.Internal.ScpiLogger.ScpiLogger

Scpi Logger interface, see *here*

Access snippet: `logger = RsOsp.utilities.logger`

property driver_version: str

Returns the instrument driver version.

property idn_string: str

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

property manufacturer: str

Returns manufacturer of the instrument.

property full_instrument_model_name: str

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

property instrument_model_name: str

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

property supported_models: List[str]

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

property instrument_firmware_version: str

Returns instrument's firmware version.

property instrument_serial_number: str

Returns instrument's serial_number.

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.

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 to keep 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.

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 bin_float_numbers_format: RsOsp.Internal.Conversions.BinFloatFormat
Sets / returns format of float numbers when transferred as binary data.

property bin_int_numbers_format: RsOsp.Internal.Conversions.BinIntFormat
Sets / returns format of integer numbers when transferred as binary data.

clear_status() → None
Clears instrument's status system, the session's I/O buffers and the instrument's error queue.

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.

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.

reset() → None
SCPI command: *RST Sends *RST command + calls the clear_status().

self_test(*timeout: Optional[int] = None*) → Tuple[int, str]
SCPI command: *TST? Performs instrument's selftest. 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 default selftest timeout is used (usually 60 secs).

is_connection_active() → bool
Returns true, if the VISA connection is active and the communication with the instrument still works.

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.

property resource_name: int
Returns the resource name used in the constructor

property opc_timeout: int
Sets / returns timeout in milliseconds for all the operations that use OPC synchronization.

property visa_timeout: int
Sets / returns visa IO timeout in milliseconds.

property data_chunk_size: int
Sets / returns the maximum size of one block transferred during write/read operations

property visa_manufacturer: int
Returns the manufacturer of the current VISA session.

process_all_commands() → None
SCPI command: *WAI Stops further commands processing until all commands sent before *WAI have been executed.

`write_str`(*cmd: str*) → None
: Writes the command to the instrument.

`write`(*cmd: str*) → None
: This method is an alias to the write_str(). Writes the command to the instrument as string.

`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_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_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.

`query_str`(*query: str*) → str
: Sends the query to the instrument and returns the response as string. The response is trimmed of any trailing LF characters and has no length limit.

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

`query_bool`(*query: str*) → bool
: Sends the query to the instrument and returns the response as boolean.

`query_int`(*query: str*) → int
: Sends the query to the instrument and returns the response as integer.

`query_float`(*query: str*) → float
: Sends the query to the instrument and returns the response as float.

`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.

`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 query_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.

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_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_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.

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!!!

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_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_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.

`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()`

`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

`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

`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 standard new RsOsp instance with new VISA session, the session gets a new thread lock. You can assign it to other RsOsp sessions in order to share one physical instrument with a multi-thread access.
- If you create new RsOsp 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()`

`assign_lock`(*lock: threading.RLock*) → None

Assigns the provided thread lock.

`clear_lock`()

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

CHAPTER

SEVEN

RSOSP LOGGER

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

`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: bool`

Sets the status of logging to the console. Default value is False.

`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.

CHAPTER

EIGHT

RSOSP EVENTS

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

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

NINE

INDEX

INDEX