RsLcx

Release 2.7.8

Rohde & Schwarz

Jul 04, 2023

CONTENTS:


LCX

CHAPTER

ONE

REVISION HISTORY

1.1 RsLcx

Rohde & Schwarz LCR Meter RsLcx instrument driver.

Basic Hello-World code:

```
from RsLcx import *

instr = RsLcx('TCPIP::192.168.2.101::hislip0')
idn = instr.query('*IDN?')
print('Hello, I am: ' + idn)
```

Check out the full documentation on ReadTheDocs.

Supported instruments: LCX

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

Documentation: https://RsLcx.readthedocs.io/

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

1.1.1 Version history

Latest release notes summary: First release for FW 2.007

Version 2.7

- First release for FW 2.007

CHAPTER

TWO

GETTING STARTED

2.1 Introduction

RsLcx 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 RsLcx example:

```
"""Getting started - how to work with RsLcx Python package.
This example performs basic RF settings on an R&S LCX instrument.
It shows the RsLcx calls and their corresponding SCPI commands.
Notice that the python RsLcx interfaces track the SCPI commands syntax."""

from RsLcx import *

# Open the session
lcx = RsLcx('TCPIP::10.102.52.44::HISLIP', False, False)
# Greetings, stranger...
print(f'Hello, I am: {lcx.utilities.idn_string}')

#   SOURce:FREQuency:FIXed 223000000
lcx.source.frequency.cw.set_value(223E6)

lcx.source.areGenerator.radar.base.set_attenuation(10)
```

(continues on next page)

(continued from previous page)

```
# Close the session
lcx.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

2.2 Installation

RsLcx 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 RsLcx`

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 `RsLcx` 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 6 step for installing the RsLcx offline:

- Download this python script (**Save target as**): rsinstrument_offline_install.py This installs all the preconditions that the RsLcx needs.
- Execute the script in your offline computer (supported is python 3.6 or newer)
- Download the RsLcx package to your computer from the pypi.org: https://pypi.org/project/RsLcx/#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\RsLcx-2.7.8.tar`

2.3 Finding Available Instruments

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

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

from RsLcx import *

# Use the instr_list string items as resource names in the RsLcx constructor
instr_list = RsLcx.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
"""
```

(continues on next page)

(continued from previous page)

```
from RsLcx 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 = RsLcx.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

2.4 Initiating Instrument Session

RsLcx 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 RsLcx object. Below, is a simple Hello World example. Different resource names are examples for different physical interfaces.

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

- Installed RsLcx Python module Version 2.7 or newer from pypi.org
- Installed VISA, for example R&S Visa 5.12 or newer
"""

from RsLcx import *

# A good practice is to assure that you have a certain minimum version installed
RsLcx.assert_minimum_version('2.7')
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
```

(continues on next page)

(continued from previous page)

```
driver = RsLcx(resource_string_1)

idn = driver.utilities.query_str('*IDN?')
print(f"\nHello, I am: '{idn}'")
print(f'RsLcx 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; RsLcx 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 = RsLcx('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 RsLcx 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 RsLcx allows you to choose which VISA to use:

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

from RsLcx import *

# Force use of the Rs Visa. For NI Visa, use the "SelectVisa='ni'"
driver = RsLcx('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}")
```

(continues on next page)

(continued from previous page)

```
# 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, RsLcx has something for you too - **no Visa installation raw LAN socket**:

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

from RsLcx import *

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

More `option_string` tokens are separated by comma:

```
driver = RsLcx('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 RsLcx objects:

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

from RsLcx import *
```

(continues on next page)

(continued from previous page)

```
driver1 = RsLcx('TCPIP::192.168.56.101::INSTR', True, True)
driver2 = RsLcx.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.

2.5 Plain SCPI Communication

After you have opened the session, you can use the instrument-specific part described in the RsLcx 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 RsLcx import *
```

(continues on next page)

(continued from previous page)

```
driver = RsLcx('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 RsLcx import *

driver = RsLcx('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 RsLcx raises an exception. Speaking of exceptions, an important feature of the RsLcx 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 RsLcx import *

driver = RsLcx('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'
```

(continues on next page)

(continued from previous page)

```
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
```

2.6 Error Checking

RsLcx 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.

2.7 Exception Handling

The base class for all the exceptions raised by the RsLcx 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 RsLcx import *

driver = None
# Try-catch for initialization. If an error occures, the ResourceError is raised
try:
    driver = RsLcx('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)

# 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...')
```

(continues on next page)

(continued from previous page)

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

2.8 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 RsLcx, 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 RsLcx 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')
```

2.9 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")
```

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

RsLcx 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:

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

from RsLcx import *
import time


def my_transfer_handler(args):
    """Function called each time a chunk of data is transferred"""
```

(continues on next page)

(continued from previous page)

```
    # 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 = RsLcx('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 RsLcx 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
```

2.11 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, RsLcx 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 RsLcx object
"""

import threading
from RsLcx import *


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


driver = RsLcx('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')

# 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 RsLcx objects with shared session
```

(continues on next page)

(continued from previous page)

```
"""

import threading
from RsLcx import *


def execute(session: RsLcx, 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 = RsLcx('TCPIP::192.168.56.101::INSTR')
driver2 = RsLcx.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')

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 RsLcx takes care of it for you. The text below describes this scenario.

Run the following example:

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

import threading
from RsLcx import *


def execute(session: RsLcx, 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 = RsLcx('TCPIP::192.168.56.101::INSTR')
driver2 = RsLcx('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()
    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.

2.12 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 RsLcx import *

driver = RsLcx('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
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.utilities.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 = RsLcx('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 RsLcx import *

driver = RsLcx('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. ---- ')

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.

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

```
driver.utilities.logger.log_to_udp = True
driver.utilities.logger.log_to_console_and_udp = True
```

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

```
driver.utilities.logger.udp_port = 49200
```

Another cool feature is logging only errors. To make this mode usefull for troubleshooting, you also want to see the circumstances which lead to the errors. Each 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 RsLcx import *

driver = RsLcx('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

THREE

ENUMS

3.1 HcopyFormat

```
# Example value:
value = enums.HcopyFormat.BMP
# All values (2x):
BMP | PNG
```

3.2 Impedance

```
# Example value:
value = enums.Impedance.HIGH
# All values (4x):
HIGH | LOW | R10 | R100
```

3.3 ImpedanceType

```
# First value:
value = enums.ImpedanceType.CPD
# Last value:
value = enums.ImpedanceType.ZTR
# All values (24x):
CPD | CPG | CPQ | CPRP | CSD | CSQ | CSRS | GB
LPD | LPG | LPQ | LPRP | LSD | LSQ | LSRS | MTD
NTD | RDC | RPB | RX | YTD | YTR | ZTD | ZTR
```

3.4 IntervalParameter

```
# Example value:
value = enums.IntervalParameter.POINts
# All values (2x):
POINts | STEPsize
```

3.5 LoggingMode

```
# Example value:
value = enums.LoggingMode.COUNt
# All values (4x):
COUNt | DURation | SPAN | UNLimited
```

3.6 MeasurementMode

```
# Example value:
value = enums.MeasurementMode.CONTinuous
# All values (2x):
CONTinuous | TRIGgered
```

3.7 MeasurementTimeMode

```
# Example value:
value = enums.MeasurementTimeMode.DEFault
# All values (4x):
DEFault | LONG | MEDium | SHORt
```

3.8 MeasurementType

```
# Example value:
value = enums.MeasurementType.C
# All values (4x):
C | L | R | T
```

3.9 MinOrMax

```
# Example value:
value = enums.MinOrMax.MAX
# All values (4x):
MAX | MAXimum | MIN | MINimum
```

3.10 SweepParameter

```
# Example value:
value = enums.SweepParameter.FREQuency
# All values (4x):
FREQuency | IBIas | VBIas | VOLTage
```

3.11 TurnRatio

```
# Example value:
value = enums.TurnRatio.N50
# All values (2x):
N50 | N500
```

3.12 UsbClass

```
# Example value:
value = enums.UsbClass.CDC
# All values (2x):
CDC | TMC
```

CHAPTER

FOUR

REPCAPS

4.1 Spot

```
# First value:
value = repcap.Spot.Nr1
# Range:
Nr1 .. Nr32
# All values (32x):
Nr1 | Nr2 | Nr3 | Nr4 | Nr5 | Nr6 | Nr7 | Nr8
Nr9 | Nr10 | Nr11 | Nr12 | Nr13 | Nr14 | Nr15 | Nr16
Nr17 | Nr18 | Nr19 | Nr20 | Nr21 | Nr22 | Nr23 | Nr24
Nr25 | Nr26 | Nr27 | Nr28 | Nr29 | Nr30 | Nr31 | Nr32
```

CHAPTER

FIVE

EXAMPLES

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

```
"""Getting started - how to work with RsLcx Python package.
This example performs basic RF settings on an R&S LCX instrument.
It shows the RsLcx calls and their corresponding SCPI commands.
Notice that the python RsLcx interfaces track the SCPI commands syntax."""

from RsLcx import *

# Open the session
lcx = RsLcx('TCPIP::10.102.52.44::HISLIP', False, False)
# Greetings, stranger...
print(f'Hello, I am: {lcx.utilities.idn_string}')

#   SOURce:FREQuency:FIXed 223000000
lcx.source.frequency.cw.set_value(223E6)

lcx.source.areGenerator.radar.base.set_attenuation(10)

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

CHAPTER

SIX

RSLCX API STRUCTURE

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

97 total commands, 18 Subgroups, 1 group commands

Initializes new RsLcx 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 = 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'`
- `LogToGlobalTarget = True` - Sets the logging target to the class-property previously set with RsLcx.set_global_logging_target() Default: `False`
- `LoggingToConsole = True` - Immediately starts logging to the console. Default: False
- `LoggingToUdp = True` - Immediately starts logging to the UDP port. Default: False
- `LoggingUdpPort = 49200` - UDP port to log to. Default: 49200

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.

classmethod clear_global_logging_relative_timestamp() → None

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

close() → None

Closes the active RsLcx session.

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

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

Parameters

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

get_aperture() → MeasurementTimeMode

```
# SCPI: APERture
value: enums.MeasurementTimeMode = driver.get_aperture()
```

Sets the measurement time mode and the acquisition time interval.

return

measurement_time_mode: Selects the basic measurement speed for one measurement. - SHORt: Sets the measurement time .015 s. - MEDium: Sets the measurement time 0.100 s. - LONG: Sets the measurement time 0.500 s. - DEFault: Uses the default setting SHORt.

classmethod get_global_logging_relative_timestamp() → datetime

Returns global common relative timestamp for log entries.

classmethod get_global_logging_target()

Returns global common target stream.

get_session_handle() → object

Returns the underlying session handle.

get_total_execution_time() → timedelta

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

get_total_time() → timedelta

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

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

Finds all the resources defined by the expression

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

Parameters

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

reset_time_statistics() → None

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

set_aperture(*measurement_time_mode: MeasurementTimeMode*) → None

```
# SCPI: APERture
driver.set_aperture(measurement_time_mode = enums.MeasurementTimeMode.DEFault)
```

Sets the measurement time mode and the acquisition time interval.

param measurement_time_mode
Selects the basic measurement speed for one measurement. - SHORt: Sets the measurement time .015 s. - MEDium: Sets the measurement time 0.100 s. - LONG: Sets the measurement time 0.500 s. - DEFault: Uses the default setting SHORt.

classmethod set_global_logging_relative_timestamp(*timestamp: datetime*) → None

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

classmethod set_global_logging_relative_timestamp_now() → None

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

classmethod set_global_logging_target(*target*) → None

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

Subgroups

6.1 Bias

SCPI Commands

```
BIAS:STATe
```

class BiasCls

Bias commands group definition. 5 total commands, 3 Subgroups, 1 group commands

get_state() → bool

```
# SCPI: BIAS:STATe
value: bool = driver.bias.get_state()
```

Activates the internal DC bias. To set the corresponding bias voltage or current value, use commands method RsLcx.Bias. Current.level, or method RsLcx.Bias.Voltage.level.

return
bias_state: No help available

set_state(*bias_state: bool*) → None

```
# SCPI: BIAS:STATe
driver.bias.set_state(bias_state = False)
```

Activates the internal DC bias. To set the corresponding bias voltage or current value, use commands method RsLcx.Bias. Current.level, or method RsLcx.Bias.Voltage.level.

param bias_state
No help available

Subgroups

6.1.1 Current

SCPI Commands

```
BIAS:CURRent:LEVel
```

class CurrentCls

Current commands group definition. 1 total commands, 0 Subgroups, 1 group commands

get_level() → float

```
# SCPI: BIAS:CURRent[:LEVel]
value: float = driver.bias.current.get_level()
```

Sets the internal bias current value. To activate the bias, use command method RsLcx.Bias.state.

return
bias_current_level: No help available

set_level(*bias_current_level: float*) → None

```
# SCPI: BIAS:CURRent[:LEVel]
driver.bias.current.set_level(bias_current_level = 1.0)
```

Sets the internal bias current value. To activate the bias, use command method RsLcx.Bias.state.

param bias_current_level
No help available

6.1.2 External

class ExternalCls

External commands group definition. 2 total commands, 2 Subgroups, 0 group commands

Subgroups

6.1.2.1 Measure

SCPI Commands

```
BIAS:EXTernal:MEASure:VOLTage
```

class MeasureCls

Measure commands group definition. 1 total commands, 0 Subgroups, 1 group commands

get_voltage() → float

```
# SCPI: BIAS:EXTernal:MEASure:VOLTage
value: float = driver.bias.external.measure.get_voltage()
```

Queries the value of the externally applied voltage.

return
voltage: No help available

6.1.2.2 Voltage

SCPI Commands

```
BIAS:EXTernal:VOLTage:STATe
```

class VoltageCls

Voltage commands group definition. 1 total commands, 0 Subgroups, 1 group commands

get_state() → bool

```
# SCPI: BIAS:EXTernal[:VOLTage][:STATe]
value: bool = driver.bias.external.voltage.get_state()
```

Activates the externally supplied bias voltage.

return
external_voltage_bias_state: No help available

set_state(*external_voltage_bias_state: bool*) → None

```
# SCPI: BIAS:EXTernal[:VOLTage][:STATe]
driver.bias.external.voltage.set_state(external_voltage_bias_state = False)
```

Activates the externally supplied bias voltage.

param external_voltage_bias_state
No help available

6.1.3 Voltage

SCPI Commands

```
BIAS:VOLTage:LEVel
```

class VoltageCls

Voltage commands group definition. 1 total commands, 0 Subgroups, 1 group commands

get_level() → float

```
# SCPI: BIAS:VOLTage[:LEVel]
value: float = driver.bias.voltage.get_level()
```

Sets the internal DC bias voltage value. To activate the bias, use command method RsLcx.Bias.state.

return
bias_voltage_level: No help available

set_level(*bias_voltage_level: float*) → None

```
# SCPI: BIAS:VOLTage[:LEVel]
driver.bias.voltage.set_level(bias_voltage_level = 1.0)
```

Sets the internal DC bias voltage value. To activate the bias, use command method RsLcx.Bias.state.

param bias_voltage_level
No help available

6.2 Correction

SCPI Commands

```
CORRection:LENGth
```

class CorrectionCls

Correction commands group definition. 13 total commands, 4 Subgroups, 1 group commands

get_length() → float

```
# SCPI: CORRection:LENGth
value: float = driver.correction.get_length()
```

Sets the length of the leads to the connected test fixture i.e. the DUT.

return
cable_length: No help available

set_length(*cable_length: float*) → None

```
# SCPI: CORRection:LENGth
driver.correction.set_length(cable_length = 1.0)
```

Sets the length of the leads to the connected test fixture i.e. the DUT.

param cable_length
No help available

Subgroups

6.2.1 Load

SCPI Commands

```
CORRection:LOAD:STATe
CORRection:LOAD:MODE
```

class LoadCls

Load commands group definition. 2 total commands, 0 Subgroups, 2 group commands

get_mode() → str

```
# SCPI: CORRection:LOAD:MODE
value: str = driver.correction.load.get_mode()
```

No command help available

return
result: No help available

get_state() → bool

```
# SCPI: CORRection:LOAD:STATe
value: bool = driver.correction.load.get_state()
```

Activates the load correction function.

return
load_correction_state: No help available

set_state(*load_correction_state: bool*) → None

```
# SCPI: CORRection:LOAD:STATe
driver.correction.load.set_state(load_correction_state = False)
```

Activates the load correction function.

param load_correction_state
No help available

6.2.2 Open

SCPI Commands

```
CORRection:OPEN:STATe
CORRection:OPEN:MODE
```

class OpenCls

Open commands group definition. 3 total commands, 1 Subgroups, 2 group commands

get_mode() → str

```
# SCPI: CORRection:OPEN:MODE
value: str = driver.correction.open.get_mode()
```

No command help available

return
result: No help available

get_state() → bool

```
# SCPI: CORRection:OPEN:STATe
value: bool = driver.correction.open.get_state()
```

Activates the open correction function.

return
open_correction_state: No help available

set_state(*open_correction_state: bool*) → None

```
# SCPI: CORRection:OPEN:STATe
driver.correction.open.set_state(open_correction_state = False)
```

Activates the open correction function.

param open_correction_state
No help available

Subgroups

6.2.2.1 Execute

SCPI Commands

```
CORRection:OPEN:EXECute
```

class ExecuteCls

Execute commands group definition. 1 total commands, 0 Subgroups, 1 group commands

set(*opc_timeout_ms: int = -1*) → None

```
# SCPI: CORRection:OPEN[:EXECute]
driver.correction.open.execute.set()
```

Executes an open correction on all frequencies.

param opc_timeout_ms
Maximum time to wait in milliseconds, valid only for this call.

6.2.3 Short

SCPI Commands

```
CORRection:SHORt:STATe
CORRection:SHORt:MODE
```

class ShortCls

Short commands group definition. 3 total commands, 1 Subgroups, 2 group commands

get_mode() → str

```
# SCPI: CORRection:SHORt:MODE
value: str = driver.correction.short.get_mode()
```

No command help available

return
result: No help available

get_state() → bool

```
# SCPI: CORRection:SHORt:STATe
value: bool = driver.correction.short.get_state()
```

Activates the short correction function.

return
short_correction_state: No help available

set_state(*short_correction_state: bool*) → None

```
# SCPI: CORRection:SHORt:STATe
driver.correction.short.set_state(short_correction_state = False)
```

Activates the short correction function.

param short_correction_state
No help available

Subgroups

6.2.3.1 Execute

SCPI Commands

```
CORRection:SHORt:EXECute
```

class ExecuteCls

Execute commands group definition. 1 total commands, 0 Subgroups, 1 group commands

set(*opc_timeout_ms: int = -1*) → None

```
# SCPI: CORRection:SHORt[:EXECute]
driver.correction.short.execute.set()
```

Executes a short correction on all frequencies.

param opc_timeout_ms
Maximum time to wait in milliseconds, valid only for this call.

6.2.4 Spot<Spot>

RepCap Settings

```
# Range: Nr1 .. Nr32
rc = driver.correction.spot.repcap_spot_get()
driver.correction.spot.repcap_spot_set(repcap.Spot.Nr1)
```

class SpotCls

Spot commands group definition. 4 total commands, 3 Subgroups, 0 group commands Repeated Capability: Spot, default value after init: Spot.Nr1

Subgroups

6.2.4.1 Load

class LoadCls

Load commands group definition. 2 total commands, 2 Subgroups, 0 group commands

Subgroups

6.2.4.1.1 Execute

SCPI Commands

```
CORRection:SPOT<Spot>:LOAD:EXECute
```

class ExecuteCls

Execute commands group definition. 1 total commands, 0 Subgroups, 1 group commands

set(*spot=Spot.Default*, *opc_timeout_ms: int = -1*) → None

```
# SCPI: CORRection:SPOT<Spot>:LOAD[:EXECute]
driver.correction.spot.load.execute.set(spot = repcap.Spot.Default)
```

Executes a load correction at a dedicated working point.

param spot
optional repeated capability selector. Default value: Nr1 (settable in the interface 'Spot')

param opc_timeout_ms
Maximum time to wait in milliseconds, valid only for this call.

6.2.4.1.2 Standard

SCPI Commands

```
CORRection:SPOT<Spot>:LOAD:STANdard
```

class StandardCls

Standard commands group definition. 1 total commands, 0 Subgroups, 1 group commands

class StandardStruct

Response structure. Fields:

- Reference_Value_For_Primary: float: Sets the primary standard value as reference, e.g. the value of a calibration resistor.
- Reference_Value_For_Secondary: float: Sets the secondary standard value.

get(*spot=Spot.Default*) → StandardStruct

```
# SCPI: CORRection:SPOT<Spot>:LOAD:STANdard
value: StandardStruct = driver.correction.spot.load.standard.get(spot = repcap.
↪Spot.Default)
```

Defines a working point for load correction. Assign the working point number and the primary and secondary reference values.

param spot
optional repeated capability selector. Default value: Nr1 (settable in the interface 'Spot')

return
structure: for return value, see the help for StandardStruct structure arguments.

set(*reference_value_for_primary: float*, *reference_value_for_secondary: float*, *spot=Spot.Default*) → None

```
# SCPI: CORRection:SPOT<Spot>:LOAD:STANdard
driver.correction.spot.load.standard.set(reference_value_for_primary = 1.0,␣
↪reference_value_for_secondary = 1.0, spot = repcap.Spot.Default)
```

Defines a working point for load correction. Assign the working point number and the primary and secondary reference values.

param reference_value_for_primary
Sets the primary standard value as reference, e.g. the value of a calibration resistor.

param reference_value_for_secondary
Sets the secondary standard value.

param spot
optional repeated capability selector. Default value: Nr1 (settable in the interface 'Spot')

6.2.4.2 Open

class OpenCls

Open commands group definition. 1 total commands, 1 Subgroups, 0 group commands

Subgroups

6.2.4.2.1 Execute

SCPI Commands

```
CORRection:SPOT<Spot>:OPEN:EXECute
```

class ExecuteCls

Execute commands group definition. 1 total commands, 0 Subgroups, 1 group commands

set(*spot=Spot.Default*, *opc_timeout_ms: int = -1*) → None

```
# SCPI: CORRection:SPOT<Spot>:OPEN[:EXECute]
driver.correction.spot.open.execute.set(spot = repcap.Spot.Default)
```

Executes an open correction at a dedicated working point.

param spot
optional repeated capability selector. Default value: Nr1 (settable in the interface 'Spot')

param opc_timeout_ms
Maximum time to wait in milliseconds, valid only for this call.

6.2.4.3 Short

class ShortCls

Short commands group definition. 1 total commands, 1 Subgroups, 0 group commands

Subgroups

6.2.4.3.1 Execute

SCPI Commands

```
CORRection:SPOT<Spot>:SHORt:EXECute
```

class ExecuteCls

Execute commands group definition. 1 total commands, 0 Subgroups, 1 group commands

set(*spot=Spot.Default*, *opc_timeout_ms: int = -1*) → None

```
# SCPI: CORRection:SPOT<Spot>:SHORt[:EXECute]
driver.correction.spot.short.execute.set(spot = repcap.Spot.Default)
```

Executes a short correction at a dedicated working point.

param spot
optional repeated capability selector. Default value: Nr1 (settable in the interface 'Spot')

param opc_timeout_ms
Maximum time to wait in milliseconds, valid only for this call.

6.3 Current

SCPI Commands

```
CURRent:LEVel
```

class CurrentCls

Current commands group definition. 1 total commands, 0 Subgroups, 1 group commands

get_level() → float

```
# SCPI: CURRent[:LEVel]
value: float = driver.current.get_level()
```

Sets the test signal current in RMS (root mean square) .

return
current_level: No help available

set_level(*current_level: float*) → None

```
# SCPI: CURRent[:LEVel]
driver.current.set_level(current_level = 1.0)
```

Sets the test signal current in RMS (root mean square) .

param current_level
No help available

6.4 Data

SCPI Commands

```
DATA:DELete
DATA:LIST
```

class DataCls

Data commands group definition. 4 total commands, 2 Subgroups, 2 group commands

delete(*file_path: str*) → None

```
# SCPI: DATA:DELete
driver.data.delete(file_path = '1')
```

Removes a file from the specified directory.

param file_path
No help available

get_list_py() → List[str]

```
# SCPI: DATA:LIST
value: List[str] = driver.data.get_list_py()
```

Queries all files in a specified directory.

return
result: No help available

Subgroups

6.4.1 Data

SCPI Commands

```
DATA:DATA
```

class DataCls

Data commands group definition. 1 total commands, 0 Subgroups, 1 group commands

get(*file_path: str*) → bytes

```
# SCPI: DATA:DATA
value: bytes = driver.data.data.get(file_path = '1')
```

Queries the contents of a file, e.g. the data of a logging file.

param file_path
No help available

return
result: No help available

6.4.2 Points

SCPI Commands

```
DATA:POINts
```

class PointsCls

Points commands group definition. 1 total commands, 0 Subgroups, 1 group commands

get(*file_path: str*) → int

```
# SCPI: DATA:POINts
value: int = driver.data.points.get(file_path = r1)
```

Queries the number of measurement readings saved in a file.

param file_path
No help available

return
count: No help available

6.5 DiMeasure

SCPI Commands

```
DIMeasure:ABORt
```

class DiMeasureCls

DiMeasure commands group definition. 8 total commands, 3 Subgroups, 1 group commands

abort() → None

```
# SCPI: DIMeasure:ABORt
driver.diMeasure.abort()
```

Stops a running dynamic impedance measurement.

abort_with_opc(*opc_timeout_ms: int = -1*) → None

```
# SCPI: DIMeasure:ABORt
driver.diMeasure.abort_with_opc()
```

Stops a running dynamic impedance measurement.

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

param opc_timeout_ms
Maximum time to wait in milliseconds, valid only for this call.

Subgroups

6.5.1 Execute

SCPI Commands

```
DIMeasure:EXECute
```

class ExecuteCls

Execute commands group definition. 1 total commands, 0 Subgroups, 1 group commands

set() → None

```
# SCPI: DIMeasure:EXECute
driver.diMeasure.execute.set()
```

Starts the dynamic impedance measurement with the selected parameters.

set_with_opc(*opc_timeout_ms: int = -1*) → None

```
# SCPI: DIMeasure:EXECute
driver.diMeasure.execute.set_with_opc()
```

Starts the dynamic impedance measurement with the selected parameters.

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

param opc_timeout_ms
Maximum time to wait in milliseconds, valid only for this call.

6.5.2 Interval

SCPI Commands

```
DIMeasure:INTerval:TYPE
DIMeasure:INTerval:STEPsize
DIMeasure:INTerval:POINts
```

class IntervalCls

Interval commands group definition. 3 total commands, 0 Subgroups, 3 group commands

get_points() → float

```
# SCPI: DIMeasure:INTerval:POINts
value: float = driver.diMeasure.interval.get_points()
```

Sets the number of measurement points within the measurement range for interval type method RsLcx.DiMeasure.Interval. typePy.

return
interval_points: No help available

get_stepsize() → float

```
# SCPI: DIMeasure:INTerval:STEPsize
value: float = driver.diMeasure.interval.get_stepsize()
```

Sets the step size within the measurement range for interval type method RsLcx.DiMeasure.Interval.typePy.

return
interval_stepsize: No help available

get_type_py() → IntervalParameter

```
# SCPI: DIMeasure:INTerval:TYPE
value: enums.IntervalParameter = driver.diMeasure.interval.get_type_py()
```

Selects the mode to determine the measurement steps within the sweep range (method RsLcx.DiMeasure.Sweep. minimum to method RsLcx.DiMeasure.Sweep.maximum) .

return
interval_parameter: - STEPsize: Measures in defined step sizes within the sweep range, set with DIMeasure:INTerval:STEPsize. - POINts: Measures in increments calculated by the number of sweep points (DIMeasure:INTerval:POINts) within the sweep range.

set_points(*interval_points: float*) → None

```
# SCPI: DIMeasure:INTerval:POINts
driver.diMeasure.interval.set_points(interval_points = 1.0)
```

Sets the number of measurement points within the measurement range for interval type method RsLcx.DiMeasure.Interval. typePy.

param interval_points
No help available

set_stepsize(*interval_stepsize: float*) → None

```
# SCPI: DIMeasure:INTerval:STEPsize
driver.diMeasure.interval.set_stepsize(interval_stepsize = 1.0)
```

Sets the step size within the measurement range for interval type method RsLcx.DiMeasure.Interval.typePy.

param interval_stepsize
No help available

set_type_py(*interval_parameter: IntervalParameter*) → None

```
# SCPI: DIMeasure:INTerval:TYPE
driver.diMeasure.interval.set_type_py(interval_parameter = enums.
↪IntervalParameter.POINts)
```

Selects the mode to determine the measurement steps within the sweep range (method RsLcx.DiMeasure.Sweep. minimum to method RsLcx.DiMeasure.Sweep.maximum) .

param interval_parameter

- STEPsize: Measures in defined step sizes within the sweep range, set with DIMeasure:INTerval:STEPsize.
- POINts: Measures in increments calculated by the number of sweep points (DIMeasure:INTerval:POINts) within the sweep range.

6.5.3 Sweep

SCPI Commands

```
DIMeasure:SWEep:PARameter
DIMeasure:SWEep:MINimum
DIMeasure:SWEep:MAXimum
```

class SweepCls

Sweep commands group definition. 3 total commands, 0 Subgroups, 3 group commands

get_maximum() → float

```
# SCPI: DIMeasure:SWEep:MAXimum
value: float = driver.diMeasure.sweep.get_maximum()
```

Sets the stop value for the selected sweep parameter. The value must be at least method RsLcx.DiMeasure.Sweep.minimum. The maximum value depends on the instrument model and installed options.

return
sweep_stop_value: No help available

get_minimum() → float

```
# SCPI: DIMeasure:SWEep:MINimum
value: float = driver.diMeasure.sweep.get_minimum()
```

Sets the start value for the selected sweep parameter. The value depends on the instrument model and installed options. The maximum value must be at least method RsLcx.DiMeasure.Sweep.maximum.

return
sweep_start_value: No help available

get_parameter() → SweepParameter

```
# SCPI: DIMeasure:SWEep:PARameter
value: enums.SweepParameter = driver.diMeasure.sweep.get_parameter()
```

Selects the measurement parameter that varies in defined steps within the sweep range (method RsLcx.DiMeasure.Sweep. minimum.

return
sweep_parameter: - VOLTage: Sweeps the test signal voltage. - FREQuency: Sweeps the test signal frequency. - VBIas: Sweeps the voltage bias. - IBIas: Sweeps the current bias.

set_maximum(*sweep_stop_value: float*) → None

```
# SCPI: DIMeasure:SWEep:MAXimum
driver.diMeasure.sweep.set_maximum(sweep_stop_value = 1.0)
```

Sets the stop value for the selected sweep parameter. The value must be at least method RsLcx.DiMeasure.Sweep.minimum. The maximum value depends on the instrument model and installed options.

param sweep_stop_value
No help available

set_minimum(*sweep_start_value: float*) → None

```
# SCPI: DIMeasure:SWEep:MINimum
driver.diMeasure.sweep.set_minimum(sweep_start_value = 1.0)
```

Sets the start value for the selected sweep parameter. The value depends on the instrument model and installed options. The maximum value must be at least method RsLcx.DiMeasure.Sweep.maximum.

param sweep_start_value
No help available

set_parameter(*sweep_parameter: SweepParameter*) → None

```
# SCPI: DIMeasure:SWEep:PARameter
driver.diMeasure.sweep.set_parameter(sweep_parameter = enums.SweepParameter.
↪FREQuency)
```

Selects the measurement parameter that varies in defined steps within the sweep range (method RsLcx.DiMeasure.Sweep. minimum.

param sweep_parameter

- VOLTage: Sweeps the test signal voltage.
- FREQuency: Sweeps the test signal frequency.
- VBIas: Sweeps the voltage bias.
- IBIas: Sweeps the current bias.

6.6 Display

SCPI Commands

```
DISPlay:BRIGhtness
```

class DisplayCls

Display commands group definition. 3 total commands, 1 Subgroups, 1 group commands

get_brightness() → float

```
# SCPI: DISPlay:BRIGhtness
value: float = driver.display.get_brightness()
```

Sets the brightness of the display.

return
display_brightness: No help available

set_brightness(*display_brightness: float*) → None

```
# SCPI: DISPlay:BRIGhtness
driver.display.set_brightness(display_brightness = 1.0)
```

Sets the brightness of the display.

param display_brightness
No help available

Subgroups

6.6.1 Window

class WindowCls

Window commands group definition. 2 total commands, 1 Subgroups, 0 group commands

Subgroups

6.6.1.1 Text

SCPI Commands

```
DISPlay:WINDow:TEXT:DATA
DISPlay:WINDow:TEXT:CLEar
```

class TextCls

Text commands group definition. 2 total commands, 0 Subgroups, 2 group commands

clear() → None

```
# SCPI: DISPlay[:WINDow]:TEXT:CLEar
driver.display.window.text.clear()
```

Closes a user defined text message on the display. To create an own message, use command method RsLcx.Display.Window.Text. data.

clear_with_opc(*opc_timeout_ms: int = -1*) → None

```
# SCPI: DISPlay[:WINDow]:TEXT:CLEar
driver.display.window.text.clear_with_opc()
```

Closes a user defined text message on the display. To create an own message, use command method RsLcx.Display.Window.Text. data.

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

param opc_timeout_ms
Maximum time to wait in milliseconds, valid only for this call.

set_data(*message: str*) → None

```
# SCPI: DISPlay[:WINDow]:TEXT[:DATA]
driver.display.window.text.set_data(message = r1)
```

Enables you to post a text message on the display. To close user defined message, use command method RsLcx.Display.Window. Text.clear.

param message
Text message for display.

6.7 Fetch

SCPI Commands

```
FETCh:IMPedance
FETCh
```

class FetchCls

Fetch commands group definition. 2 total commands, 0 Subgroups, 2 group commands

get_impedance() → List[float]

```
# SCPI: FETCh:IMPedance
value: List[float] = driver.fetch.get_impedance()
```

Queries the most recent valid values of the measured impedance. If no valid measurement values are available, the reponse reports error code -230.

return
impedance: No help available

get_value() → List[float]

```
# SCPI: FETCh
value: List[float] = driver.fetch.get_value()
```

Queries the most recent valid values for measurement pair 2. If no valid measurement values are available, the reponse reports error code -230.

return
results: No help available

6.8 Frequency

SCPI Commands

```
FREQuency:CW
```

class FrequencyCls

Frequency commands group definition. 1 total commands, 0 Subgroups, 1 group commands

get_cw() → float

```
# SCPI: FREQuency[:CW]
value: float = driver.frequency.get_cw()
```

Sets the frequency of the test signal.

return
test_signal_frequency: No help available

set_cw(*test_signal_frequency: float*) → None

```
# SCPI: FREQuency[:CW]
driver.frequency.set_cw(test_signal_frequency = 1.0)
```

Sets the frequency of the test signal.

param test_signal_frequency
No help available

6.9 Function

class FunctionCls

Function commands group definition. 7 total commands, 3 Subgroups, 0 group commands

Subgroups

6.9.1 Impedance

SCPI Commands

```
FUNCtion:IMPedance:TYPE
FUNCtion:IMPedance:SOURce
```

class ImpedanceCls

Impedance commands group definition. 5 total commands, 1 Subgroups, 2 group commands

get_source() → Impedance

```
# SCPI: FUNCtion:IMPedance:SOURce
value: enums.Impedance = driver.function.impedance.get_source()
```

Selects the output impedance for the measurement.

return
impedance: - LOW | R10: Sets 10 output impedance. - HIGH | R100: Sets 100 output impedance.

get_type_py() → ImpedanceType

```
# SCPI: FUNCtion:IMPedance[:TYPE]
value: enums.ImpedanceType = driver.function.impedance.get_type_py()
```

Selects the impedance parameter for the measurement corresponding to the measurement type, see method RsLcx.Function. Measurement.typePy.

return
impedance_type: - CPD | CPQ | CPG | CPRP | CSD | CSQ | CSRS: Capacitive measurement type: Cp (parallel capacitance) , Cs (serial capacitance) , D (dissipation factor) , Q (quality factor) , G (conductance) , Rp (parallel resistance) , Rs (serial resistance) - LPD | LPQ | LPG | LPRP | LPRDc | LSD | LSQ | LSRS | LSRDc: Inductive measurement type: Lp (parallel inductance) , Ls (serial inductance) , D (dissipation factor) , Q (quality factor) , G (conductance) , Rp (parallel resistance) , Rs (serial resistance) , RDc (direct current resistance) - RX | RPB | RDC | MTD | NTD | ZTD | ZTR | GB | YTD | YTR: Resistance measurement type: R (resistance) , X impedance, Rp (parallel resistance) , RDC (direct current resistance) , B (susceptance) , M (mutual inductance) , N (transformer ratio) , Z (impedance) , G (conductance) , Y (admittance) , TD (phase angle degree) , TR (phase angle rad)

set_source(*impedance: Impedance*) → None

```
# SCPI: FUNCtion:IMPedance:SOURce
driver.function.impedance.set_source(impedance = enums.Impedance.HIGH)
```

Selects the output impedance for the measurement.

param impedance

- LOW | R10: Sets 10 output impedance.
- HIGH | R100: Sets 100 output impedance.

set_type_py(*impedance_type: ImpedanceType*) → None

```
# SCPI: FUNCtion:IMPedance[:TYPE]
driver.function.impedance.set_type_py(impedance_type = enums.ImpedanceType.CPD)
```

Selects the impedance parameter for the measurement corresponding to the measurement type, see method RsLcx.Function. Measurement.typePy.

param impedance_type

- CPD | CPQ | CPG | CPRP | CSD | CSQ | CSRS: Capacitive measurement type: Cp (parallel capacitance) , Cs (serial capacitance) , D (dissipation factor) , Q (quality factor) , G (conductance) , Rp (parallel resistance) , Rs (serial resistance)

- LPD | LPQ | LPG | LPRP | LPRDc | LSD | LSQ | LSRS | LSRDc: Inductive measurement type: Lp (parallel inductance) , Ls (serial inductance) , D (dissipation factor) , Q (quality factor) , G (conductance) , Rp (parallel resistance) , Rs (serial resistance) , RDc (direct current resistance)
- RX | RPB | RDC | MTD | NTD | ZTD | ZTR | GB | YTD | YTR: Resistance measurement type: R (resistance) , X impedance, Rp (parallel resistance) , RDC (direct current resistance) , B (susceptance) , M (mutual inductance) , N (transformer ratio) , Z (impedance) , G (conductance) , Y (admittance) , TD (phase angle degree) , TR (phase angle rad)

Subgroups

6.9.1.1 Range

SCPI Commands

```
FUNCtion:IMPedance:RANGe:AUTO
FUNCtion:IMPedance:RANGe:HOLD
FUNCtion:IMPedance:RANGe:VALue
```

class RangeCls

Range commands group definition. 3 total commands, 0 Subgroups, 3 group commands

get_auto() → bool

```
# SCPI: FUNCtion:IMPedance:RANGe:AUTO
value: bool = driver.function.impedance.range.get_auto()
```

Activates automatic impedance range selection. To set the impedance range manually, use command method RsLcx.Function. Impedance.Range.value.

return
auto_range: No help available

get_hold() → bool

```
# SCPI: FUNCtion:IMPedance:RANGe:HOLD
value: bool = driver.function.impedance.range.get_hold()
```

Freezes the set impedance measurement range.

return
locks_selected_range: No help available

get_value() → float

```
# SCPI: FUNCtion:IMPedance:RANGe[:VALue]
value: float = driver.function.impedance.range.get_value()
```

Sets the impedance range value. For setting the parameter manually, disable auto selection with method RsLcx.Function. Impedance.Range.auto.

return
range_py: No help available

set_auto(*auto_range: bool*) → None

```
# SCPI: FUNCtion:IMPedance:RANGe:AUTO
driver.function.impedance.range.set_auto(auto_range = False)
```

Activates automatic impedance range selection. To set the impedance range manually, use command method RsLcx.Function. Impedance.Range.value.

param auto_range
No help available

set_hold(*locks_selected_range: bool*) → None

```
# SCPI: FUNCtion:IMPedance:RANGe:HOLD
driver.function.impedance.range.set_hold(locks_selected_range = False)
```

Freezes the set impedance measurement range.

param locks_selected_range
No help available

set_value(*range_py: float*) → None

```
# SCPI: FUNCtion:IMPedance:RANGe[:VALue]
driver.function.impedance.range.set_value(range_py = 1.0)
```

Sets the impedance range value. For setting the parameter manually, disable auto selection with method RsLcx.Function. Impedance.Range.auto.

param range_py
No help available

6.9.2 Measurement

SCPI Commands

```
FUNCtion:MEASurement:TYPE
```

class MeasurementCls

Measurement commands group definition. 1 total commands, 0 Subgroups, 1 group commands

get_type_py() → MeasurementType

```
# SCPI: FUNCtion:MEASurement:TYPE
value: enums.MeasurementType = driver.function.measurement.get_type_py()
```

Selects the measurement function.

return
measurement_type: - L: Impedance measurement. - C: Capacitance measurement. - R: Resistance measurement. - T: Transformer measurement.

set_type_py(*measurement_type: MeasurementType*) → None

```
# SCPI: FUNCtion:MEASurement:TYPE
driver.function.measurement.set_type_py(measurement_type = enums.
↪MeasurementType.C)
```

Selects the measurement function.

param measurement_type

- L: Impedance measurement.
- C: Capacitance measurement.
- R: Resistance measurement.
- T: Transformer measurement.

6.9.3 Transformer

class TransformerCls

Transformer commands group definition. 1 total commands, 1 Subgroups, 0 group commands

Subgroups

6.9.3.1 Range

SCPI Commands

```
FUNCtion:TRANsformer:RANGe:TYPE
```

class RangeCls

Range commands group definition. 1 total commands, 0 Subgroups, 1 group commands

get_type_py() → TurnRatio

```
# SCPI: FUNCtion:TRANsformer:RANGe[:TYPE]
value: enums.TurnRatio = driver.function.transformer.range.get_type_py()
```

Selects the impedance range for transformer measurement.

return
turn_ratio: No help available

set_type_py(*turn_ratio: TurnRatio*) → None

```
# SCPI: FUNCtion:TRANsformer:RANGe[:TYPE]
driver.function.transformer.range.set_type_py(turn_ratio = enums.TurnRatio.N50)
```

Selects the impedance range for transformer measurement.

param turn_ratio
No help available

6.10 Handler

SCPI Commands

```
HANDler:STATe
```

class HandlerCls

Handler commands group definition. 5 total commands, 2 Subgroups, 1 group commands

get_state() → bool

```
# SCPI: HANDler[:STATe]
value: bool = driver.handler.get_state()
```

Activates the binning measurement.

return
handler_state: No help available

set_state(*handler_state: bool*) → None

```
# SCPI: HANDler[:STATe]
driver.handler.set_state(handler_state = False)
```

Activates the binning measurement.

param handler_state
No help available

Subgroups

6.10.1 Bin

class BinCls

Bin commands group definition. 3 total commands, 1 Subgroups, 0 group commands

Subgroups

6.10.1.1 Statistic

SCPI Commands

```
HANDler:BIN:STATistic:COUNt
HANDler:BIN:STATistic:RESet
HANDler:BIN:STATistic
```

class StatisticCls

Statistic commands group definition. 3 total commands, 0 Subgroups, 3 group commands

get_count() → int

```
# SCPI: HANDler:BIN:STATistic:COUNt
value: int = driver.handler.bin.statistic.get_count()
```

Queries the total number of samples measured since reset The query returns the sum of all counts.

return
count: No help available

get_value() → List[int]

```
# SCPI: HANDler:BIN:STATistic
value: List[int] = driver.handler.bin.statistic.get_value()
```

Queries the number of samples counted in the bins. The query returns 8 integer values.

return
statictics: No help available

reset() → None

```
# SCPI: HANDler:BIN:STATistic:RESet
driver.handler.bin.statistic.reset()
```

Resets the evaluated binning measurement statistics.

reset_with_opc(*opc_timeout_ms: int = -1*) → None

```
# SCPI: HANDler:BIN:STATistic:RESet
driver.handler.bin.statistic.reset_with_opc()
```

Resets the evaluated binning measurement statistics.

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

param opc_timeout_ms
Maximum time to wait in milliseconds, valid only for this call.

6.10.2 Config

SCPI Commands

```
HANDler:CONFig:PATH
```

class ConfigCls

Config commands group definition. 1 total commands, 0 Subgroups, 1 group commands

get_path() → str

```
# SCPI: HANDler:CONFig:PATH
value: str = driver.handler.config.get_path()
```

Uploads the binning configuration file.

return
file_path: No help available

set_path(*file_path: str*) → None

```
# SCPI: HANDler:CONFig:PATH
driver.handler.config.set_path(file_path = '1')
```

Uploads the binning configuration file.

param file_path
No help available

6.11 HardCopy

SCPI Commands

```
HCOPy:DATA
```

class HardCopyCls

HardCopy commands group definition. 4 total commands, 2 Subgroups, 1 group commands

get_data() → bytes

```
# SCPI: HCOPy:DATA
value: bytes = driver.hardCopy.get_data()
```

No command help available

return
result: No help available

Subgroups

6.11.1 FormatPy

SCPI Commands

```
HCOPy:FORMat
```

class FormatPyCls

FormatPy commands group definition. 1 total commands, 0 Subgroups, 1 group commands

get(*format_py: HcopyFormat*) → HcopyFormat

```
# SCPI: HCOPy:FORMat
value: enums.HcopyFormat = driver.hardCopy.formatPy.get(format_py = enums.
↪HcopyFormat.BMP)
```

No command help available

param format_py
No help available

return
format_py: No help available

set(*format_py: HcopyFormat*) → None

```
# SCPI: HCOPy:FORMat
driver.hardCopy.formatPy.set(format_py = enums.HcopyFormat.BMP)
```

No command help available

param format_py
No help available

6.11.2 Size

SCPI Commands

```
HCOPy:SIZE:X
HCOPy:SIZE:Y
```

class SizeCls

Size commands group definition. 2 total commands, 0 Subgroups, 2 group commands

get_x() → int

```
# SCPI: HCOPy:SIZE:X
value: int = driver.hardCopy.size.get_x()
```

No command help available

return
size_x: No help available

get_y() → int

```
# SCPI: HCOPy:SIZE:Y
value: int = driver.hardCopy.size.get_y()
```

No command help available

return
size_y: No help available

6.12 Initiate

class InitiateCls

Initiate commands group definition. 1 total commands, 1 Subgroups, 0 group commands

Subgroups

6.12.1 Immediate

SCPI Commands

```
INITiate:IMMediate
```

class ImmediateCls

Immediate commands group definition. 1 total commands, 0 Subgroups, 1 group commands

set() → None

```
# SCPI: INITiate[:IMMediate]
driver.initiate.immediate.set()
```

Starts a new measurement. In manual trigger mode, the command triggers a single measurement cycle. When completed, the R&S LCX waits for the next trigger event.

set_with_opc(*opc_timeout_ms: int = -1*) → None

```
# SCPI: INITiate[:IMMediate]
driver.initiate.immediate.set_with_opc()
```

Starts a new measurement. In manual trigger mode, the command triggers a single measurement cycle. When completed, the R&S LCX waits for the next trigger event.

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

param opc_timeout_ms
Maximum time to wait in milliseconds, valid only for this call.

6.13 Interfaces

class InterfacesCls

Interfaces commands group definition. 1 total commands, 1 Subgroups, 0 group commands

Subgroups

6.13.1 Usb

SCPI Commands

```
INTerfaces:USB:CLASs
```

class UsbCls

Usb commands group definition. 1 total commands, 0 Subgroups, 1 group commands

get_class_py() → UsbClass

```
# SCPI: INTerfaces:USB:CLASs
value: enums.UsbClass = driver.interfaces.usb.get_class_py()
```

Selects the USB communication class.

return
usb_class: - CDC: Uses the virtual communication port protocol, that enables you to emulate serial ports over USB. - TMC: Uses the protocol for communication with USB devices.

set_class_py(*usb_class: UsbClass*) → None

```
# SCPI: INTerfaces:USB:CLASs
driver.interfaces.usb.set_class_py(usb_class = enums.UsbClass.CDC)
```

Selects the USB communication class.

param usb_class

- CDC: Uses the virtual communication port protocol, that enables you to emulate serial ports over USB.
- TMC: Uses the protocol for communication with USB devices.

6.14 Log

SCPI Commands

```
LOG:STATe
LOG:MODE
LOG:FNAMe
LOG:STIMe
```

class LogCls

Log commands group definition. 7 total commands, 3 Subgroups, 4 group commands

class StimeStruct

Structure for setting input parameters. Fields:

- Year: int: Four-digit number, including the century and millennium information.
- Month: int: No parameter help available
- Day: int: No parameter help available
- Hour: int: No parameter help available
- Minute: int: No parameter help available
- Second: int: No parameter help available

get_fname() → str

```
# SCPI: LOG:FNAMe
value: str = driver.log.get_fname()
```

Sets the file name and path for the storing the data recorded during data logging. The query returns the file name and path. You can query the information also when data logging is running.

return
logging_file_name: String with the directory and filename.

get_mode() → LoggingMode

```
# SCPI: LOG:MODE
value: enums.LoggingMode = driver.log.get_mode()
```

Selects the data logging mode.

return
logging_mode: - UNLimited: No specified limit of measurement readings. - COUNt: Determines the number of measurement readings. - DURation: Sets a time interval between the measurement readings. - SPAN: Defines start time and time span for the measurement readings.

get_state() → bool

```
# SCPI: LOG[:STATe]
value: bool = driver.log.get_state()
```

Activates the data logging function.

return
logging_state: No help available

get_stime() → StimeStruct

```
# SCPI: LOG:STIMe
value: StimeStruct = driver.log.get_stime()
```

Sets the logging start time.

return
structure: for return value, see the help for StimeStruct structure arguments.

set_fname(*logging_file_name: str*) → None

```
# SCPI: LOG:FNAMe
driver.log.set_fname(logging_file_name = '1')
```

Sets the file name and path for the storing the data recorded during data logging. The query returns the file name and path. You can query the information also when data logging is running.

param logging_file_name
String with the directory and filename.

set_mode(*logging_mode: LoggingMode*) → None

```
# SCPI: LOG:MODE
driver.log.set_mode(logging_mode = enums.LoggingMode.COUNt)
```

Selects the data logging mode.

param logging_mode

- UNLimited: No specified limit of measurement readings.

- COUNt: Determines the number of measurement readings.
- DURation: Sets a time interval between the measurement readings.
- SPAN: Defines start time and time span for the measurement readings.

set_state(*logging_state: bool*) → None

```
# SCPI: LOG[:STATe]
driver.log.set_state(logging_state = False)
```

Activates the data logging function.

param logging_state
No help available

set_stime(*value: StimeStruct*) → None

```
# SCPI: LOG:STIMe
structure = driver.log.StimeStruct()
structure.Year: int = 1
structure.Month: int = 1
structure.Day: int = 1
structure.Hour: int = 1
structure.Minute: int = 1
structure.Second: int = 1
driver.log.set_stime(value = structure)
```

Sets the logging start time.

param value
see the help for StimeStruct structure arguments.

Subgroups

6.14.1 Count

SCPI Commands

```
LOG:COUNt
```

class CountCls

Count commands group definition. 1 total commands, 0 Subgroups, 1 group commands

get() → float

```
# SCPI: LOG:COUNt
value: float = driver.log.count.get()
```

Sets the number of measurement readings in count mode. To set the mode, use the command method RsLcx.Log. mode to be captured.

return
sample_count: No help available

set(*sample_count: float*, *return_min_or_max: MinOrMax = None*) → None

```
# SCPI: LOG:COUNt
driver.log.count.set(sample_count = 1.0, return_min_or_max = enums.MinOrMax.MAX)
```

Sets the number of measurement readings in count mode. To set the mode, use the command method RsLcx.Log. mode to be captured.

param sample_count
No help available

param return_min_or_max
No help available

6.14.2 Duration

SCPI Commands

```
LOG:DURation
```

class DurationCls

Duration commands group definition. 1 total commands, 0 Subgroups, 1 group commands

get() → float

```
# SCPI: LOG:DURation
value: float = driver.log.duration.get()
```

Defines the duration of logging for the measurement in span and duration mode. To set the mode, use the command method RsLcx.Log.mode.

return
logging_duration: Numeric value in seconds.

set(*logging_duration: float*, *return_min_or_max: MinOrMax = None*) → None

```
# SCPI: LOG:DURation
driver.log.duration.set(logging_duration = 1.0, return_min_or_max = enums.
↪MinOrMax.MAX)
```

Defines the duration of logging for the measurement in span and duration mode. To set the mode, use the command method RsLcx.Log.mode.

param logging_duration
Numeric value in seconds.

param return_min_or_max
No help available

6.14.3 Interval

SCPI Commands

```
LOG:INTerval
```

class IntervalCls

Interval commands group definition. 1 total commands, 0 Subgroups, 1 group commands

get() → float

```
# SCPI: LOG:INTerval
value: float = driver.log.interval.get()
```

Selects the logging measurement interval. The measurement interval describes the time between the recorded measurements.

return
sample_interval: Numeric value in seconds. - 0: Logs a new measurement.

set(*sample_interval: float*, *return_min_or_max: MinOrMax = None*) → None

```
# SCPI: LOG:INTerval
driver.log.interval.set(sample_interval = 1.0, return_min_or_max = enums.
↪MinOrMax.MAX)
```

Selects the logging measurement interval. The measurement interval describes the time between the recorded measurements.

param sample_interval
Numeric value in seconds. - 0: Logs a new measurement.

param return_min_or_max
No help available

6.15 Measure

SCPI Commands

```
MEASure:VOLTage
MEASure:CURRent
MEASure:MODE
MEASure:ACCuracy
```

class MeasureCls

Measure commands group definition. 5 total commands, 1 Subgroups, 4 group commands

class Result

Structure for reading output parameters. Fields:

- Percent: float: No parameter help available
- Degrees: float: No parameter help available

get_accuracy() → Result

```
# SCPI: MEASure:ACCuracy
value: Result = driver.measure.get_accuracy()
```

Queries the accuracy of the last measurement. The R&S LCX returns the accuracy of the impedance (|Z|) in percent, and the phase angle () in degrees.

return
structure: for return value, see the help for Result structure arguments.

get_current() → float

```
# SCPI: MEASure:CURRent
value: float = driver.measure.get_current()
```

Queries the current value following next in the measurement.

return
current: No help available

get_mode() → MeasurementMode

```
# SCPI: MEASure:MODE
value: enums.MeasurementMode = driver.measure.get_mode()
```

Selects whether the R&S LCX starts and continues a measurement, or starts on initiated trigger events.

return
measurement_mode: - CONTinuous: Restarts the measurement automatically after a measurement cycle has been completed. - TRIGgered: Starts a measurement cycle initiated by a trigger signal.To delay the measurement start to a certain extent, use command MEASure:TRIGger:DELay.

get_voltage() → float

```
# SCPI: MEASure:VOLTage
value: float = driver.measure.get_voltage()
```

Queries the voltage value following next in the measurement.

return
voltage: No help available

set_mode(*measurement_mode: MeasurementMode*) → None

```
# SCPI: MEASure:MODE
driver.measure.set_mode(measurement_mode = enums.MeasurementMode.CONTinuous)
```

Selects whether the R&S LCX starts and continues a measurement, or starts on initiated trigger events.

param measurement_mode

- CONTinuous: Restarts the measurement automatically after a measurement cycle has been completed.
- TRIGgered: Starts a measurement cycle initiated by a trigger signal.To delay the measurement start to a certain extent, use command MEASure:TRIGger:DELay.

Subgroups

6.15.1 Trigger

SCPI Commands

```
MEASure:TRIGger:DELay
```

class TriggerCls

Trigger commands group definition. 1 total commands, 0 Subgroups, 1 group commands

get_delay() → float

```
# SCPI: MEASure:TRIGger:DELay
value: float = driver.measure.trigger.get_delay()
```

Sets a delay time that elapses after a trigger event before the measurement starts.

return
trigger_delay: No help available

set_delay(*trigger_delay: float*) → None

```
# SCPI: MEASure:TRIGger:DELay
driver.measure.trigger.set_delay(trigger_delay = 1.0)
```

Sets a delay time that elapses after a trigger event before the measurement starts.

param trigger_delay
No help available

6.16 Read

SCPI Commands

```
READ:IMPedance
READ
```

class ReadCls

Read commands group definition. 2 total commands, 0 Subgroups, 2 group commands

get_impedance() → List[float]

```
# SCPI: READ:IMPedance
value: List[float] = driver.read.get_impedance()
```

Queries the impedance measurement results.

return
impedance: No help available

get_value() → List[float]

```
# SCPI: READ
value: List[float] = driver.read.get_value()
```

Queries the measurement results for measurement pair 2.

return
results: No help available

6.17 System

SCPI Commands

```
SYSTem:UPTime
```

class SystemCls

System commands group definition. 26 total commands, 11 Subgroups, 1 group commands

get_up_time() → str

```
# SCPI: SYSTem:UPTime
value: str = driver.system.get_up_time()
```

Queries the up time of the operating system.

return
result: No help available

Subgroups

6.17.1 Beeper

class BeeperCls

Beeper commands group definition. 4 total commands, 2 Subgroups, 0 group commands

Subgroups

6.17.1.1 Complete

SCPI Commands

```
SYSTem:BEEPer:COMPlete:STATe
```

class CompleteCls

Complete commands group definition. 2 total commands, 1 Subgroups, 1 group commands

get_state() → bool

```
# SCPI: SYSTem:BEEPer[:COMPlete]:STATe
value: bool = driver.system.beeper.complete.get_state()
```

Activates the R&S LCX to create an acoustic signal on a completed operation. The query returns the current state.

return
enable: No help available

set_state(*enable: bool*) → None

```
# SCPI: SYSTem:BEEPer[:COMPlete]:STATe
driver.system.beeper.complete.set_state(enable = False)
```

Activates the R&S LCX to create an acoustic signal on a completed operation. The query returns the current state.

param enable
No help available

Subgroups

6.17.1.1.1 Immediate

SCPI Commands

```
SYSTem:BEEPer:COMPlete:IMMediate
```

class ImmediateCls

Immediate commands group definition. 1 total commands, 0 Subgroups, 1 group commands

set() → None

```
# SCPI: SYSTem:BEEPer[:COMPlete][:IMMediate]
driver.system.beeper.complete.immediate.set()
```

Activates that the R&S LCX issues a beep after operation complete immediately. The query returns the current state.

set_with_opc(*opc_timeout_ms: int = -1*) → None

```
# SCPI: SYSTem:BEEPer[:COMPlete][:IMMediate]
driver.system.beeper.complete.immediate.set_with_opc()
```

Activates that the R&S LCX issues a beep after operation complete immediately. The query returns the current state.

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

param opc_timeout_ms
Maximum time to wait in milliseconds, valid only for this call.

6.17.1.2 WarningPy

SCPI Commands

```
SYSTem:BEEPer:WARNing:STATe
```

class WarningPyCls

WarningPy commands group definition. 2 total commands, 1 Subgroups, 1 group commands

get_state() → bool

```
# SCPI: SYSTem:BEEPer:WARNing:STATe
value: bool = driver.system.beeper.warningPy.get_state()
```

Activates the R&S LCX to create an acoustic signal on errors and warnings. The query returns the current state.

return
enable: No help available

set_state(*enable: bool*) → None

```
# SCPI: SYSTem:BEEPer:WARNing:STATe
driver.system.beeper.warningPy.set_state(enable = False)
```

Activates the R&S LCX to create an acoustic signal on errors and warnings. The query returns the current state.

param enable
No help available

Subgroups

6.17.1.2.1 Immediate

SCPI Commands

```
SYSTem:BEEPer:WARNing:IMMediate
```

class ImmediateCls

Immediate commands group definition. 1 total commands, 0 Subgroups, 1 group commands

set() → None

```
# SCPI: SYSTem:BEEPer:WARNing[:IMMediate]
driver.system.beeper.warningPy.immediate.set()
```

Activates that the R&S LCX issues a beep on error or warning immediately.

set_with_opc(*opc_timeout_ms: int = -1*) → None

```
# SCPI: SYSTem:BEEPer:WARNing[:IMMediate]
driver.system.beeper.warningPy.immediate.set_with_opc()
```

Activates that the R&S LCX issues a beep on error or warning immediately.

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

param opc_timeout_ms
Maximum time to wait in milliseconds, valid only for this call.

6.17.2 Communicate

class CommunicateCls

Communicate commands group definition. 12 total commands, 2 Subgroups, 0 group commands

Subgroups

6.17.2.1 Lan

SCPI Commands

```
SYSTem:COMMunicate:LAN:DHCP
SYSTem:COMMunicate:LAN:ADDRess
SYSTem:COMMunicate:LAN:SMASk
SYSTem:COMMunicate:LAN:DGATeway
SYSTem:COMMunicate:LAN:HOSTname
SYSTem:COMMunicate:LAN:MAC
SYSTem:COMMunicate:LAN:RESet
SYSTem:COMMunicate:LAN:EDITed
```

class LanCls

Lan commands group definition. 10 total commands, 2 Subgroups, 8 group commands

get_address() → str

```
# SCPI: SYSTem:COMMunicate:LAN:ADDRess
value: str = driver.system.communicate.lan.get_address()
```

Sets the IP address.

return
ip_address: No help available

get_dgateway() → str

```
# SCPI: SYSTem:COMMunicate:LAN:DGATeway
value: str = driver.system.communicate.lan.get_dgateway()
```

Sets the IP address of the default gateway.

return
gateway: No help available

get_dhcp() → bool

```
# SCPI: SYSTem:COMMunicate:LAN:DHCP
value: bool = driver.system.communicate.lan.get_dhcp()
```

No command help available

return
enable: No help available

get_edited() → bool

```
# SCPI: SYSTem:COMMunicate:LAN:EDITed
value: bool = driver.system.communicate.lan.get_edited()
```

No command help available

return
result: No help available

get_hostname() → str

```
# SCPI: SYSTem:COMMunicate:LAN:HOSTname
value: str = driver.system.communicate.lan.get_hostname()
```

Sets an individual hostname for the R&S LCX.

return
device_hostname: No help available

get_mac() → str

```
# SCPI: SYSTem:COMMunicate:LAN:MAC
value: str = driver.system.communicate.lan.get_mac()
```

Queries the MAC address of the network.

return
result: No help available

get_smask() → str

```
# SCPI: SYSTem:COMMunicate:LAN:SMASk
value: str = driver.system.communicate.lan.get_smask()
```

Sets the subnet mask.

return
subnet_mask: No help available

reset() → None

```
# SCPI: SYSTem:COMMunicate:LAN:RESet
driver.system.communicate.lan.reset()
```

Terminates the network configuration and restarts the network.

reset_with_opc(*opc_timeout_ms: int = -1*) → None

```
# SCPI: SYSTem:COMMunicate:LAN:RESet
driver.system.communicate.lan.reset_with_opc()
```

Terminates the network configuration and restarts the network.

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

param opc_timeout_ms
Maximum time to wait in milliseconds, valid only for this call.

set_address(*ip_address: str*) → None

```
# SCPI: SYSTem:COMMunicate:LAN:ADDRess
driver.system.communicate.lan.set_address(ip_address = '1')
```

Sets the IP address.

param ip_address
No help available

set_dgateway(*gateway: str*) → None

```
# SCPI: SYSTem:COMMunicate:LAN:DGATeway
driver.system.communicate.lan.set_dgateway(gateway = '1')
```

Sets the IP address of the default gateway.

param gateway
No help available

set_dhcp(*enable: bool*) → None

```
# SCPI: SYSTem:COMMunicate:LAN:DHCP
driver.system.communicate.lan.set_dhcp(enable = False)
```

No command help available

param enable
No help available

set_hostname(*device_hostname: str*) → None

```
# SCPI: SYSTem:COMMunicate:LAN:HOSTname
driver.system.communicate.lan.set_hostname(device_hostname = '1')
```

Sets an individual hostname for the R&S LCX.

param device_hostname
No help available

set_smask(*subnet_mask: str*) → None

```
# SCPI: SYSTem:COMMunicate:LAN:SMASk
driver.system.communicate.lan.set_smask(subnet_mask = '1')
```

Sets the subnet mask.

param subnet_mask
No help available

Subgroups

6.17.2.1.1 Apply

SCPI Commands

```
SYSTem:COMMunicate:LAN:APPLy
```

class ApplyCls

Apply commands group definition. 1 total commands, 0 Subgroups, 1 group commands

set() → None

```
# SCPI: SYSTem:COMMunicate:LAN:APPLy
driver.system.communicate.lan.apply.set()
```

Assigns and confirms the settings.

set_with_opc(*opc_timeout_ms: int = -1*) → None

```
# SCPI: SYSTem:COMMunicate:LAN:APPLy
driver.system.communicate.lan.apply.set_with_opc()
```

Assigns and confirms the settings.

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

param opc_timeout_ms
Maximum time to wait in milliseconds, valid only for this call.

6.17.2.1.2 Discard

SCPI Commands

```
SYSTem:COMMunicate:LAN:DISCard
```

class DiscardCls

Discard commands group definition. 1 total commands, 0 Subgroups, 1 group commands

set() → None

```
# SCPI: SYSTem:COMMunicate:LAN:DISCard
driver.system.communicate.lan.discard.set()
```

Removes the LAN configuration.

set_with_opc(*opc_timeout_ms: int = -1*) → None

```
# SCPI: SYSTem:COMMunicate:LAN:DISCard
driver.system.communicate.lan.discard.set_with_opc()
```

Removes the LAN configuration.

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

param opc_timeout_ms
Maximum time to wait in milliseconds, valid only for this call.

6.17.2.2 Network

class NetworkCls

Network commands group definition. 2 total commands, 1 Subgroups, 0 group commands

Subgroups

6.17.2.2.1 Vnc

SCPI Commands

```
SYSTem:COMMunicate:NETWork:VNC:STATe
SYSTem:COMMunicate:NETWork:VNC:PORT
```

class VncCls

Vnc commands group definition. 2 total commands, 0 Subgroups, 2 group commands

get_port() → int

```
# SCPI: SYSTem:COMMunicate:NETWork:VNC:PORT
value: int = driver.system.communicate.network.vnc.get_port()
```

Sets the VNC port address.

return
port: No help available

get_state() → bool

```
# SCPI: SYSTem:COMMunicate:NETWork:VNC[:STATe]
value: bool = driver.system.communicate.network.vnc.get_state()
```

Activates the VNC interface for remote access.

return
enable: No help available

set_port(*port: int*) → None

```
# SCPI: SYSTem:COMMunicate:NETWork:VNC:PORT
driver.system.communicate.network.vnc.set_port(port = 1)
```

Sets the VNC port address.

param port
No help available

set_state(*enable: bool*) → None

```
# SCPI: SYSTem:COMMunicate:NETWork:VNC[:STATe]
driver.system.communicate.network.vnc.set_state(enable = False)
```

Activates the VNC interface for remote access.

param enable
No help available

6.17.3 Date

SCPI Commands

```
SYSTem:DATE
```

class DateCls

Date commands group definition. 1 total commands, 0 Subgroups, 1 group commands

class DateStruct

Response structure. Fields:

- Year: float: Sets the year.
- Month: float: Sets the month.
- Day: float: Sets the day.

get() → DateStruct

```
# SCPI: SYSTem:DATE
value: DateStruct = driver.system.date.get()
```

Sets or queries the date for the instrument-internal calendar.

return
structure: for return value, see the help for DateStruct structure arguments.

set(*year: float*, *month: float*, *day: float*) → None

```
# SCPI: SYSTem:DATE
driver.system.date.set(year = 1.0, month = 1.0, day = 1.0)
```

Sets or queries the date for the instrument-internal calendar.

param year
Sets the year.

param month
Sets the month.

param day
Sets the day.

6.17.4 Hw

SCPI Commands

```
SYSTem:HW:VERSion
```

class HwCls

Hw commands group definition. 1 total commands, 0 Subgroups, 1 group commands

get_version() → str

```
# SCPI: SYSTem:HW:VERSion
value: str = driver.system.hw.get_version()
```

Queries the hardware version of the instrument.

return
result: No help available

6.17.5 Key

SCPI Commands

```
SYSTem:KEY:BRIGhtness
```

class KeyCls

Key commands group definition. 1 total commands, 0 Subgroups, 1 group commands

get_brightness() → float

```
# SCPI: SYSTem:KEY:BRIGhtness
value: float = driver.system.key.get_brightness()
```

Sets the brightness of the front panel keys.

return
front_key_brightness: No help available

set_brightness(*front_key_brightness: float*) → None

```
# SCPI: SYSTem:KEY:BRIGhtness
driver.system.key.set_brightness(front_key_brightness = 1.0)
```

Sets the brightness of the front panel keys.

param front_key_brightness
No help available

6.17.6 Local

SCPI Commands

```
SYSTem:LOCal
```

class LocalCls

Local commands group definition. 1 total commands, 0 Subgroups, 1 group commands

set() → None

```
# SCPI: SYSTem:LOCal
driver.system.local.set()
```

Enables manual operation, i.e. unlocks front panel control. To lock manual control, use command method RsLcx.System. RwLock.set.

set_with_opc(*opc_timeout_ms: int = -1*) → None

```
# SCPI: SYSTem:LOCal
driver.system.local.set_with_opc()
```

Enables manual operation, i.e. unlocks front panel control. To lock manual control, use command method RsLcx.System. RwLock.set.

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

param opc_timeout_ms
Maximum time to wait in milliseconds, valid only for this call.

6.17.7 Remote

SCPI Commands

```
SYSTem:REMote
```

class RemoteCls

Remote commands group definition. 1 total commands, 0 Subgroups, 1 group commands

set() → None

```
# SCPI: SYSTem:REMote
driver.system.remote.set()
```

Activates remote control. The R&S LCX switches to remote state, and locks all front panel controls. You can control the R&S LCX remotely. Sending a command sets the instrument to remote state, indicated by the white SCPI icon in the status bar. To return to manual control, use command method RsLcx.System.Local.set.

set_with_opc(*opc_timeout_ms: int = -1*) → None

```
# SCPI: SYSTem:REMote
driver.system.remote.set_with_opc()
```

Activates remote control. The R&S LCX switches to remote state, and locks all front panel controls. You can control the R&S LCX remotely. Sending a command sets the instrument to remote state, indicated by the white SCPI icon in the status bar. To return to manual control, use command method RsLcx.System.Local.set.

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

param opc_timeout_ms
Maximum time to wait in milliseconds, valid only for this call.

6.17.8 Restart

SCPI Commands

```
SYSTem:RESTart
```

class RestartCls

Restart commands group definition. 1 total commands, 0 Subgroups, 1 group commands

set() → None

```
# SCPI: SYSTem:RESTart
driver.system.restart.set()
```

Restarts the instrument without restarting the operating system.

set_with_opc(*opc_timeout_ms: int = -1*) → None

```
# SCPI: SYSTem:RESTart
driver.system.restart.set_with_opc()
```

Restarts the instrument without restarting the operating system.

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

param opc_timeout_ms
Maximum time to wait in milliseconds, valid only for this call.

6.17.9 RwLock

SCPI Commands

```
SYSTem:RWLock
```

class RwLockCls

RwLock commands group definition. 1 total commands, 0 Subgroups, 1 group commands

set() → None

```
# SCPI: SYSTem:RWLock
driver.system.rwLock.set()
```

Locks all front panel controls, i.e. manual operation. To unlock the front panel control, use command method RsLcx.System. Local.set.

set_with_opc(*opc_timeout_ms: int = -1*) → None

```
# SCPI: SYSTem:RWLock
driver.system.rwLock.set_with_opc()
```

Locks all front panel controls, i.e. manual operation. To unlock the front panel control, use command method RsLcx.System. Local.set.

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

param opc_timeout_ms
Maximum time to wait in milliseconds, valid only for this call.

6.17.10 Setting

class SettingCls

Setting commands group definition. 1 total commands, 1 Subgroups, 0 group commands

Subgroups

6.17.10.1 Default

SCPI Commands

```
SYSTem:SETTing:DEFault:SAVE
```

class DefaultCls

Default commands group definition. 1 total commands, 0 Subgroups, 1 group commands

save(*file_path: str = None*) → None

```
# SCPI: SYSTem:SETTing:DEFault:SAVE
driver.system.setting.default.save(file_path = '1')
```

Saves the current instrument settings in a file with defined filename. To recall an instrument configuration, use command *RCL.

param file_path
No help available

6.17.11 Time

SCPI Commands

```
SYSTem:TIME
```

class TimeCls

Time commands group definition. 1 total commands, 0 Subgroups, 1 group commands

class TimeStruct

Response structure. Fields:

- Hour: int: No parameter help available
- Minute: int: No parameter help available
- Second: int: No parameter help available

get() → TimeStruct

```
# SCPI: SYSTem:TIME
value: TimeStruct = driver.system.time.get()
```

Sets or queries the time for the instrument-internal clock. The R&S LCX indicates the time in the status bar.

return
structure: for return value, see the help for TimeStruct structure arguments.

set(*hour: int*, *minute: int*, *second: int*) → None

```
# SCPI: SYSTem:TIME
driver.system.time.set(hour = 1, minute = 1, second = 1)
```

Sets or queries the time for the instrument-internal clock. The R&S LCX indicates the time in the status bar.

param hour
No help available

param minute
No help available

param second
No help available

6.18 Voltage

SCPI Commands

```
VOLTage:LEVel
```

class VoltageCls

Voltage commands group definition. 1 total commands, 0 Subgroups, 1 group commands

get_level() → float

```
# SCPI: VOLTage[:LEVel]
value: float = driver.voltage.get_level()
```

Sets the test signal voltage in RMS (root mean square) .

return
voltage_level: - numeric: Sets the value. - MIN | MINimum: Queries the lower limit of the signal level. - MAX | MAXimum: Queries the upper limit of the signal level. - DEF | DEFault: Queries the signal level the instrument sets by default.

set_level(*voltage_level: float*) → None

```
# SCPI: VOLTage[:LEVel]
driver.voltage.set_level(voltage_level = 1.0)
```

Sets the test signal voltage in RMS (root mean square) .

param voltage_level

- numeric: Sets the value.
- MIN | MINimum: Queries the lower limit of the signal level.
- MAX | MAXimum: Queries the upper limit of the signal level.
- DEF | DEFault: Queries the signal level the instrument sets by default.

CHAPTER

SEVEN

RSLCX UTILITIES

class Utilities

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

Access snippet: `utils = RsLcx.utilities`

property logger: *ScpiLogger*

Scpi Logger interface, see *here*

Access snippet: `logger = RsLcx.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 encoding: str

Returns string<=>bytes encoding of the session.

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: BinFloatFormat

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

property bin_int_numbers_format: BinIntFormat

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

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().

default_instrument_setup() → None

Custom steps performed at the init and at the reset().

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

SCPI command: *TST? Performs instrument's self-test. Returns tuple (code:int, message: str). Code 0 means the self-test passed. You can define the custom timeout in milliseconds. If you do not define it, the 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: 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: 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: 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: int = None*) → None

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

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

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

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

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

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

Sends a OPC-synced query and reads 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). 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: int = None*) → List[int]

Sends a OPC-synced query and reads 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). 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: 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() → RLock

Returns the thread lock for the current session.

By default:

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

sync_from(*source: Utilities*) → None

Synchronises these Utils with the source.

CHAPTER

EIGHT

RSLCX 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: bool = None*, *udp_log: bool = None*) → None

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

get_logging_target()

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

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

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

log_to_console

Returns logging to console status.

log_to_udp

Returns logging to UDP status.

log_to_console_and_udp

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

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

Method for logging the raw string without any formatting.

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

Method for logging one error entry.

set_relative_timestamp(*timestamp: datetime*) → None

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

set_relative_timestamp_now() → None

Sets the relative timestamp to the current time.

get_relative_timestamp() → datetime

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

clear_relative_timestamp() → None

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

flush() → None

Flush all the entries.

log_status_check_ok

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

clear_cached_entries() → None

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

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

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

restore_format_string() → None

Restores the original format string and the line divider to LF

abbreviated_max_len_ascii: int

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

abbreviated_max_len_bin: int

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

abbreviated_max_len_list: int

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

bin_line_block_size: int

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

udp_port

Returns udp logging port.

target_auto_flushing

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

CHAPTER

NINE

RSLCX 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`

sync_from(*source: Events*) → None

Synchronises these Events with the source.

CHAPTER

TEN

INDEX

INDEX

I

L

M

R

S

T

U

V