RsCmwGprfGen

Release 4.0.140.55

Rohde & Schwarz

Apr 16, 2024

CONTENTS:


CMW

CHAPTER

ONE

REVISION HISTORY

1.1 RsCmwGprfGen

Rohde & Schwarz CMW GPRF Generator RsCmwGprfGen instrument driver.

Basic Hello-World code:

```
from RsCmwGprfGen import *

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

Supported instruments: CMW500, CMW270, CMW280, CMW100

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

Documentation: https://RsCmwGprfGen.readthedocs.io/

Examples: https://github.com/Rohde-Schwarz/Examples/

1.1.1 Version history

Release Notes:

Latest release notes summary: Update for FW 4.0.140

Version 4.0.140

- Update for FW 4.0.140

Version 3.8.xx2

- Fixed several misspelled arguments and command headers

Version 3.8.xx1

- Bluetooth and WLAN update for FW versions 3.8.xxx

Version 3.7.xx8

- Added documentation on ReadTheDocs

Version 3.7.xx7

- Added 3G measurement subsystems RsCmwGsmMeas, RsCmwCdma2kMeas, RsCmwEvdoMeas, RsCmwWcdmaMeas
- Added new data types for commands accepting numbers or ON/OFF:
- int or bool
- float or bool

Version 3.7.xx6

- Added new UDF integer number recognition

Version 3.7.xx5

- Added RsCmwDau

Version 3.7.xx4

- Fixed several interface names
- New release for CMW Base 3.7.90
- New release for CMW Bluetooth 3.7.90

Version 3.7.xx3

- Second release of the CMW python drivers packet
- New core component RsInstrument
- Previously, the groups starting with CATalog: e.g. 'CATalog:SIGNaling:TOPology:PLMN' were reordered to 'SIGNaling:TOPology:PLMN:CATALOG' give more contextual meaning to the method/property name. This is now reverted back, since it was hard to find the desired functionality.
- Reorganized Utilities interface to sub-groups

Version 3.7.xx2

- Fixed some misspeling errors
- Changed enum and repCap types names
- All the assemblies are signed with Rohde & Schwarz signature

Version 1.0.0.0

- First released version

CHAPTER

TWO

GETTING STARTED

2.1 Introduction

RsCmwGprfGen 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 RsCmwBase example:

```
""" Example on how to use the python RsCmw auto-generated instrument driver showing:
- usage of basic properties of the cmw_base object
- basic concept of setting commands and repcaps: DISPlay:WINDow<n>:SELect
- cmw_xxx drivers reliability interface usage
"""

from RsCmwBase import *  # install from pypi.org

RsCmwBase.assert_minimum_version('3.7.90.38')
cmw_base = RsCmwBase('TCPIP::10.112.1.116::INSTR', True, False)
print(f'CMW Base IND: {cmw_base.utilities.idn_string}')
print(f'CMW Instrument options:\n{",".join(cmw_base.utilities.instrument_options)}')
cmw_base.utilities.visa_timeout = 5000

# Sends OPC after each command
cmw_base.utilities.opc_query_after_write = False
```

(continues on next page)

(continued from previous page)

```
# Checks for syst:err? after each command / query
cmw_base.utilities.instrument_status_checking = True

# DISPlay:WINDow<n>:SELect
cmw_base.display.window.select.set(repcap.Window.Win1)
cmw_base.display.window.repcap_window_set(repcap.Window.Win2)
cmw_base.display.window.select.set()

# Self-test
self_test = cmw_base.utilities.self_test()
print(f'CMW self-test result: {self_test} - {"Passed" if self_test[0] == 0 else "Failed"}
↪"')

# Driver's Interface reliability offers a convenient way of reacting on the return value␣
↪Reliability Indicator
cmw_base.reliability.ExceptionOnError = True


# Callback to use for the reliability indicator update event
def my_reliability_handler(event_args: ReliabilityEventArgs):
        print(f'Base Reliability updated.\nContext: {event_args.context}\nMessage:
↪{event_args.message}')


# We register a callback for each change in the reliability indicator
cmw_base.reliability.on_update_handler = my_reliability_handler

# You can obtain the last value of the returned reliability
print(f"\nReliability last value: {cmw_base.reliability.last_value}, context '{cmw_base.
↪reliability.last_context}', message: {cmw_base.reliability.last_message}")

# Reference Frequency Source
cmw_base.system.reference.frequency.set_source(enums.SourceIntExt.INTernal)

# Close the session
cmw_base.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

RsCmwGprfGen 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 RsCmwGprfGen`

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 `RsCmwGprfGen` 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 RsCmwGprfGen offline:

- Download this python script (**Save target as**): rsinstrument_offline_install.py This installs all the preconditions that the RsCmwGprfGen needs.
- Execute the script in your offline computer (supported is python 3.6 or newer)
- Download the RsCmwGprfGen package to your computer from the pypi.org: https://pypi.org/project/RsCmwGprfGen/#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\RsCmwGprfGen-4.0.140.55.tar`

2.3 Finding Available Instruments

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

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

from RsCmwGprfGen import *

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

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

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

from RsCmwGprfGen 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 = RsCmwGprfGen.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

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

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

- Installed RsCmwGprfGen Python module Version 4.0.140 or newer from pypi.org
- Installed VISA, for example R&S Visa 5.12 or newer
"""

from RsCmwGprfGen import *

# A good practice is to assure that you have a certain minimum version installed
RsCmwGprfGen.assert_minimum_version('4.0.140')
resource_string_1 = 'TCPIP::192.168.2.101::INSTR'  # Standard LAN connection (also␣
↪called VXI-11)
resource_string_2 = 'TCPIP::192.168.2.101::hislip0'  # Hi-Speed LAN connection - see␣
↪1MA208
resource_string_3 = 'GPIB::20::INSTR'  # GPIB Connection
resource_string_4 = 'USB::0x0AAD::0x0119::022019943::INSTR'  # USB-TMC (Test and␣
↪Measurement Class)

# Initializing the session
driver = RsCmwGprfGen(resource_string_1)

idn = driver.utilities.query_str('*IDN?')
print(f"\nHello, I am: '{idn}'")
print(f'RsCmwGprfGen 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 2023.

Do not care about specialty of each session kind; RsCmwGprfGen 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 = RsCmwGprfGen('TCPIP::192.168.56.101::hislip0', id_query=True, reset=True)
```

- Setting id_query to True (default is True) checks, whether your instrument can be used with the RsCmwGprfGen 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 RsCmwGprfGen allows you to choose which VISA to use:

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

from RsCmwGprfGen import *

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

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

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

No VISA Session

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

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

from RsCmwGprfGen import *

driver = RsCmwGprfGen('TCPIP::192.168.56.101::5025::SOCKET', True, True, "SelectVisa=
↪'socket'")
print(f'Visa manufacturer: {driver.utilities.visa_manufacturer}')
```

(continues on next page)

(continued from previous page)

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

More `option_string` tokens are separated by comma:

```
driver = RsCmwGprfGen('TCPIP::192.168.56.101::hislip0', 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 RsCmwGprfGen objects:

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

from RsCmwGprfGen import *

driver1 = RsCmwGprfGen('TCPIP::192.168.56.101::INSTR', True, True)
driver2 = RsCmwGprfGen.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 RsCmwGprfGen 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 RsCmwGprfGen import *

driver = RsCmwGprfGen('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 RsCmwGprfGen import *

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

(continues on next page)

(continued from previous page)

```
# 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 RsCmwGprfGen raises an exception. Speaking of exceptions, an important feature of the RsCmwGprfGen 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 RsCmwGprfGen import *

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

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

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

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

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

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

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

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

2.6 Error Checking

RsCmwGprfGen 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 RsCmwGprfGen 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 RsCmwGprfGen import *

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

(continues on next page)

(continued from previous page)

```
    driver = RsCmwGprfGen('TCPIP::10.112.1.179::hislip0')
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...')

except RsInstrException as e:
    # RsInstrException is a base class for all the RsCmwGprfGen exceptions
    print(e.args[0])
    print('Some other RsCmwGprfGen 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 RsCmwGprfGen, 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 RsCmwGprfGen 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 RsCmwGprfGen has this covered. And, this feature is quite universal - not just for big files transfer, but for any data in both directions.

RsCmwGprfGen 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 RsCmwGprfGen import *
import time


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

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

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


driver = RsCmwGprfGen('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 RsCmwGprfGen 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, RsCmwGprfGen 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 RsCmwGprfGen object
"""

import threading
from RsCmwGprfGen import *


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


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

(continues on next page)

(continued from previous page)

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

driver.close()
```

Shared instrument session, accessed from multiple threads

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

Try the following example:

```
"""
Multiple threads are accessing two RsCmwGprfGen objects with shared session
"""

import threading
from RsCmwGprfGen import *


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

(continues on next page)

(continued from previous page)

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

Run the following example:

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

import threading
from RsCmwGprfGen import *


def execute(session: RsCmwGprfGen, 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 = RsCmwGprfGen('TCPIP::192.168.56.101::INSTR')
driver2 = RsCmwGprfGen('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,))
```

(continues on next page)

(continued from previous page)

```
    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 RsCmwGprfGen import *

driver = RsCmwGprfGen('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
```

(continues on next page)

(continued from previous page)

```
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 = RsCmwGprfGen('TCPIP::192.168.56.101::hislip0', 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 RsCmwGprfGen import *

driver = RsCmwGprfGen('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
```

(continues on next page)

(continued from previous page)

```
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 RsCmwGprfGen import *

driver = RsCmwGprfGen('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')
```

(continues on next page)

(continued from previous page)

```
# 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 ArbFile

```
# Example value:
value = enums.ArbFile.ABSPath
# All values (3x):
ABSPath | DEF | TRUTh
```

3.2 ArbSegmentsMode

```
# Example value:
value = enums.ArbSegmentsMode.AUTO
# All values (3x):
AUTO | CONTinuous | CSEamless
```

3.3 BasebandMode

```
# Example value:
value = enums.BasebandMode.ARB
# All values (3x):
ARB | CW | DTONe
```

3.4 GeneratorState

```
# Example value:
value = enums.GeneratorState.ADJusted
# All values (8x):
ADJusted | AUTonomous | COUPled | INValid | OFF | ON | PENDing | RDY
```

3.5 IncTransition

```
# Example value:
value = enums.IncTransition.IMMediate
# All values (6x):
IMMediate | RMARker | WMA1 | WMA2 | WMA3 | WMA4
```

3.6 InstrumentType

```
# Example value:
value = enums.InstrumentType.PROTocol
# All values (2x):
PROTocol | SIGNaling
```

3.7 ListIncrement

```
# Example value:
value = enums.ListIncrement.ACYCles
# All values (5x):
ACYCles | DTIMe | MEASurement | TRIGger | USER
```

3.8 ParameterSetMode

```
# Example value:
value = enums.ParameterSetMode.GLOBal
# All values (2x):
GLOBal | LIST
```

3.9 Range

```
# Example value:
value = enums.Range.FULL
# All values (2x):
FULL | SUB
```

3.10 RepeatMode

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

3.11 Scenario

```
# Example value:
value = enums.Scenario.IQOut
# All values (3x):
IQOut | NAV | SALone
```

3.12 SignalSlope

```
# Example value:
value = enums.SignalSlope.FEDGe
# All values (2x):
FEDGe | REDGe
```

3.13 TargetMainState

```
# Example value:
value = enums.TargetMainState.OFF
# All values (3x):
OFF | RUN | STOP
```

3.14 TargetSyncState

```
# Example value:
value = enums.TargetSyncState.ADJusted
# All values (2x):
ADJusted | PENDing
```

3.15 TransferMode

```
# Example value:
value = enums.TransferMode.ENABlemode
# All values (2x):
ENABlemode | REQuestmode
```

3.16 TxConnector

```
# First value:
value = enums.TxConnector.I12O
# Last value:
value = enums.TxConnector.RH18
# All values (86x):
I12O | I14O | I16O | I18O | I22O | I24O | I26O | I28O
I32O | I34O | I36O | I38O | I42O | I44O | I46O | I48O
IFO1 | IFO2 | IFO3 | IFO4 | IFO5 | IFO6 | IQ2O | IQ4O
IQ6O | IQ8O | R10D | R118 | R1183 | R1184 | R11C | R11D
R11O | R11O3 | R11O4 | R12C | R12D | R13C | R13O | R14C
R214 | R218 | R21C | R21O | R22C | R23C | R23O | R24C
R258 | R318 | R31C | R31O | R32C | R33C | R33O | R34C
R418 | R41C | R41O | R42C | R43C | R43O | R44C | RA18
RB14 | RB18 | RC18 | RD18 | RE18 | RF18 | RF1C | RF1O
RF2C | RF3C | RF3O | RF4C | RF5C | RF6C | RF7C | RF8C
RF9C | RFAC | RFAO | RFBC | RG18 | RH18
```

3.17 TxConnectorBench

```
# First value:
value = enums.TxConnectorBench.R118
# Last value:
value = enums.TxConnectorBench.RH18
# All values (15x):
R118 | R214 | R218 | R258 | R318 | R418 | RA18 | RB14
RB18 | RC18 | RD18 | RE18 | RF18 | RG18 | RH18
```

3.18 TxConnectorCmws

```
# First value:
value = enums.TxConnectorCmws.R11
# Last value:
value = enums.TxConnectorCmws.RH8
# All values (96x):
R11 | R12 | R13 | R14 | R15 | R16 | R17 | R18
R21 | R22 | R23 | R24 | R25 | R26 | R27 | R28
```

(continues on next page)

(continued from previous page)

```
R31 | R32 | R33 | R34 | R35 | R36 | R37 | R38
R41 | R42 | R43 | R44 | R45 | R46 | R47 | R48
RA1 | RA2 | RA3 | RA4 | RA5 | RA6 | RA7 | RA8
RB1 | RB2 | RB3 | RB4 | RB5 | RB6 | RB7 | RB8
RC1 | RC2 | RC3 | RC4 | RC5 | RC6 | RC7 | RC8
RD1 | RD2 | RD3 | RD4 | RD5 | RD6 | RD7 | RD8
RE1 | RE2 | RE3 | RE4 | RE5 | RE6 | RE7 | RE8
RF1 | RF2 | RF3 | RF4 | RF5 | RF6 | RF7 | RF8
RG1 | RG2 | RG3 | RG4 | RG5 | RG6 | RG7 | RG8
RH1 | RH2 | RH3 | RH4 | RH5 | RH6 | RH7 | RH8
```

3.19 TxConverter

```
# First value:
value = enums.TxConverter.ITX1
# Last value:
value = enums.TxConverter.TX44
# All values (40x):
ITX1 | ITX11 | ITX12 | ITX13 | ITX14 | ITX2 | ITX21 | ITX22
ITX23 | ITX24 | ITX3 | ITX31 | ITX32 | ITX33 | ITX34 | ITX4
ITX41 | ITX42 | ITX43 | ITX44 | TX1 | TX11 | TX12 | TX13
TX14 | TX2 | TX21 | TX22 | TX23 | TX24 | TX3 | TX31
TX32 | TX33 | TX34 | TX4 | TX41 | TX42 | TX43 | TX44
```

3.20 YesNoStatus

```
# Example value:
value = enums.YesNoStatus.NO
# All values (2x):
NO | YES
```

CHAPTER

FOUR

REPCAPS

4.1 Instance (Global)

```
# Setting:
driver.repcap_instance_set(repcap.Instance.Inst1)
# Range:
Inst1 .. Inst32
# All values (32x):
Inst1 | Inst2 | Inst3 | Inst4 | Inst5 | Inst6 | Inst7 | Inst8
Inst9 | Inst10 | Inst11 | Inst12 | Inst13 | Inst14 | Inst15 | Inst16
Inst17 | Inst18 | Inst19 | Inst20 | Inst21 | Inst22 | Inst23 | Inst24
Inst25 | Inst26 | Inst27 | Inst28 | Inst29 | Inst30 | Inst31 | Inst32
```

4.2 Bench

```
# First value:
value = repcap.Bench.Nr1
# Range:
Nr1 .. Nr20
# All values (20x):
Nr1 | Nr2 | Nr3 | Nr4 | Nr5 | Nr6 | Nr7 | Nr8
Nr9 | Nr10 | Nr11 | Nr12 | Nr13 | Nr14 | Nr15 | Nr16
Nr17 | Nr18 | Nr19 | Nr20
```

4.3 FrequencySource

```
# First value:
value = repcap.FrequencySource.Src1
# Values (2x):
Src1 | Src2
```

4.4 LevelSource

```
# First value:
value = repcap.LevelSource.Src1
# Values (2x):
Src1 | Src2
```

4.5 Marker

```
# First value:
value = repcap.Marker.Nr1
# Values (4x):
Nr1 | Nr2 | Nr3 | Nr4
```

CHAPTER

FIVE

EXAMPLES

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

```
""" Example on how to use the python RsCmw auto-generated instrument driver showing:
- usage of basic properties of the cmw_base object
- basic concept of setting commands and repcaps: DISPlay:WINDow<n>:SELect
- cmw_xxx drivers reliability interface usage
"""

from RsCmwBase import *  # install from pypi.org

RsCmwBase.assert_minimum_version('3.7.90.38')
cmw_base = RsCmwBase('TCPIP::10.112.1.116::INSTR', True, False)
print(f'CMW Base IND: {cmw_base.utilities.idn_string}')
print(f'CMW Instrument options:\n{",".join(cmw_base.utilities.instrument_options)}')
cmw_base.utilities.visa_timeout = 5000

# Sends OPC after each command
cmw_base.utilities.opc_query_after_write = False

# Checks for syst:err? after each command / query
cmw_base.utilities.instrument_status_checking = True

# DISPlay:WINDow<n>:SELect
cmw_base.display.window.select.set(repcap.Window.Win1)
cmw_base.display.window.repcap_window_set(repcap.Window.Win2)
cmw_base.display.window.select.set()

# Self-test
self_test = cmw_base.utilities.self_test()
print(f'CMW self-test result: {self_test} - {"Passed" if self_test[0] == 0 else "Failed"}
↪"')

# Driver's Interface reliability offers a convenient way of reacting on the return value␣
↪Reliability Indicator
cmw_base.reliability.ExceptionOnError = True


# Callback to use for the reliability indicator update event
def my_reliability_handler(event_args: ReliabilityEventArgs):
        print(f'Base Reliability updated.\nContext: {event_args.context}\nMessage:
```

(continues on next page)

(continued from previous page)

```
↪{event_args.message}')


# We register a callback for each change in the reliability indicator
cmw_base.reliability.on_update_handler = my_reliability_handler

# You can obtain the last value of the returned reliability
print(f"\nReliability last value: {cmw_base.reliability.last_value}, context '{cmw_base.
↪reliability.last_context}', message: {cmw_base.reliability.last_message}")

# Reference Frequency Source
cmw_base.system.reference.frequency.set_source(enums.SourceIntExt.INTernal)

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

CHAPTER

SIX

RSCMWGPRFGEN API STRUCTURE

Global RepCaps

```
driver = RsCmwGprfGen('TCPIP::192.168.2.101::hislip0')
# Instance range: Inst1 .. Inst32
rc = driver.repcap_instance_get()
driver.repcap_instance_set(repcap.Instance.Inst1)
```

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

193 total commands, 4 Subgroups, 0 group commands

Initializes new RsCmwGprfGen 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`
- `ScpiQuotes = double'. - for SCPI commands, you can define how strings are quoted. With single or double quotes. Possible values: single | double | {char}. Default: ``single`
- `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 RsCmwGprfGen.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 RsCmwGprfGen session.

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

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

Parameters

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

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()

restore_all_repcaps_to_default() → None

Sets all the Group and Global repcaps to their initial values

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 Configure

SCPI Command :

```
CONFigure:GPRF:GENerator<Instance>:TYPE
```

class ConfigureCls

Configure commands group definition. 6 total commands, 2 Subgroups, 1 group commands

get_type_py() → InstrumentType

```
# SCPI: CONFigure:GPRF:GENerator<Instance>:TYPE
value: enums.InstrumentType = driver.configure.get_type_py()
```

No command help available

return
instrument_type: No help available

set_type_py(*instrument_type: InstrumentType*) → None

```
# SCPI: CONFigure:GPRF:GENerator<Instance>:TYPE
driver.configure.set_type_py(instrument_type = enums.InstrumentType.PROTocol)
```

No command help available

param instrument_type
No help available

Cloning the Group

```
# Create a clone of the original group, that exists independently
group2 = driver.configure.clone()
```

Subgroups

6.1.1 SingleCmw

class SingleCmwCls

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

Cloning the Group

```
# Create a clone of the original group, that exists independently
group2 = driver.configure.singleCmw.clone()
```

Subgroups

6.1.1.1 Usage

class UsageCls

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

Cloning the Group

```
# Create a clone of the original group, that exists independently
group2 = driver.configure.singleCmw.usage.clone()
```

Subgroups

6.1.1.1.1 Tx

SCPI Command :

```
CONFigure:GPRF:GENerator<Instance>:CMWS:USAGe:TX
```

class TxCls

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

get(*tx_connector: TxConnectorCmws*) → bool

```
# SCPI: CONFigure:GPRF:GENerator<Instance>:CMWS:USAGe:TX
value: bool = driver.configure.singleCmw.usage.tx.get(tx_connector = enums.
↪TxConnectorCmws.R11)
```

No command help available

param tx_connector
No help available

return
usage: No help available

set(*tx_connector: TxConnectorCmws*, *usage: bool*) → None

```
# SCPI: CONFigure:GPRF:GENerator<Instance>:CMWS:USAGe:TX
driver.configure.singleCmw.usage.tx.set(tx_connector = enums.TxConnectorCmws.
↪R11, usage = False)
```

No command help available

param tx_connector
No help available

param usage
No help available

Cloning the Group

```
# Create a clone of the original group, that exists independently
group2 = driver.configure.singleCmw.usage.tx.clone()
```

Subgroups

6.1.1.1.1.1 All

SCPI Command :

```
CONFigure:GPRF:GENerator<Instance>:CMWS:USAGe:TX:ALL
```

class AllCls

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

get(*tx_connector_bench: TxConnectorBench*) → List[bool]

```
# SCPI: CONFigure:GPRF:GENerator<Instance>:CMWS:USAGe:TX:ALL
value: List[bool] = driver.configure.singleCmw.usage.tx.all.get(tx_connector_
↪bench = enums.TxConnectorBench.R118)
```

No command help available

param tx_connector_bench
No help available

return
usage: No help available

set(*tx_connector_bench: TxConnectorBench*, *usage: List[bool]*) → None

```
# SCPI: CONFigure:GPRF:GENerator<Instance>:CMWS:USAGe:TX:ALL
driver.configure.singleCmw.usage.tx.all.set(tx_connector_bench = enums.
↪TxConnectorBench.R118, usage = [True, False, True])
```

No command help available

param tx_connector_bench
No help available

param usage
No help available

6.1.2 Spath

SCPI Command :

```
CONFigure:GPRF:GENerator<Instance>:SPATh:BCSWitch
```

class SpathCls

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

get_bc_switch() → bool

```
# SCPI: CONFigure:GPRF:GENerator<Instance>:SPATh:BCSWitch
value: bool = driver.configure.spath.get_bc_switch()
```

No command help available

return
connect_switch: No help available

set_bc_switch(*connect_switch: bool*) → None

```
# SCPI: CONFigure:GPRF:GENerator<Instance>:SPATh:BCSWitch
driver.configure.spath.set_bc_switch(connect_switch = False)
```

No command help available

param connect_switch
No help available

Cloning the Group

```
# Create a clone of the original group, that exists independently
group2 = driver.configure.spath.clone()
```

Subgroups

6.1.2.1 Usage

SCPI Command :

```
CONFigure:GPRF:GENerator<Instance>:SPATh:USAGe
```

class UsageCls

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

get_value() → List[bool]

```
# SCPI: CONFigure:GPRF:GENerator<Instance>:SPATh:USAGe
value: List[bool] = driver.configure.spath.usage.get_value()
```

No command help available

return
enable: No help available

set_value(*enable: List[bool]*) → None

```
# SCPI: CONFigure:GPRF:GENerator<Instance>:SPATh:USAGe
driver.configure.spath.usage.set_value(enable = [True, False, True])
```

No command help available

param enable
No help available

Cloning the Group

```
# Create a clone of the original group, that exists independently
group2 = driver.configure.spath.usage.clone()
```

Subgroups

6.1.2.1.1 Bench<Bench>

RepCap Settings

```
# Range: Nr1 .. Nr20
rc = driver.configure.spath.usage.bench.repcap_bench_get()
driver.configure.spath.usage.bench.repcap_bench_set(repcap.Bench.Nr1)
```

SCPI Command :

```
CONFigure:GPRF:GENerator<Instance>:SPATh:USAGe:BENCh<nr>
```

class BenchCls

Bench commands group definition. 1 total commands, 0 Subgroups, 1 group commands Repeated Capability: Bench, default value after init: Bench.Nr1

get(*bench=Bench.Default*) → List[bool]

```
# SCPI: CONFigure:GPRF:GENerator<Instance>:SPATh:USAGe:BENCh<nr>
value: List[bool] = driver.configure.spath.usage.bench.get(bench = repcap.Bench.
↪Default)
```

No command help available

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

return
enable: No help available

set(*enable: List[bool]*, *bench=Bench.Default*) → None

```
# SCPI: CONFigure:GPRF:GENerator<Instance>:SPATh:USAGe:BENCh<nr>
driver.configure.spath.usage.bench.set(enable = [True, False, True], bench =␣
↪repcap.Bench.Default)
```

No command help available

param enable
No help available

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

Cloning the Group

```
# Create a clone of the original group, that exists independently
group2 = driver.configure.spath.usage.bench.clone()
```

6.2 Route

SCPI Command :

```
ROUTe:GPRF:GENerator<Instance>
```

class RouteCls

Route commands group definition. 4 total commands, 1 Subgroups, 1 group commands

class ValueStruct

Structure for reading output parameters. Fields:

- Scenario: enums.Scenario: No parameter help available
- Master: str: No parameter help available
- Tx_Connector: enums.TxConnector: No parameter help available
- Rf_Converter: enums.TxConverter: No parameter help available

get_value() → ValueStruct

```
# SCPI: ROUTe:GPRF:GENerator<Instance>
value: ValueStruct = driver.route.get_value()
```

No command help available

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

Cloning the Group

```
# Create a clone of the original group, that exists independently
group2 = driver.route.clone()
```

Subgroups

6.2.1 Scenario

SCPI Command :

```
ROUTe:GPRF:GENerator<Instance>:SCENario
```

class ScenarioCls

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

get_value() → Scenario

```
# SCPI: ROUTe:GPRF:GENerator<Instance>:SCENario
value: enums.Scenario = driver.route.scenario.get_value()
```

No command help available

return
scenario: No help available

Cloning the Group

```
# Create a clone of the original group, that exists independently
group2 = driver.route.scenario.clone()
```

Subgroups

6.2.1.1 IqOut

SCPI Command :

```
ROUTe:GPRF:GENerator<Instance>:SCENario:IQOut
```

class IqOutCls

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

class IqOutStruct

Response structure. Fields:

- Tx_Connector: enums.TxConnector: No parameter help available
- Tx_Converter: enums.TxConverter: No parameter help available

get() → IqOutStruct

```
# SCPI: ROUTe:GPRF:GENerator<Instance>:SCENario:IQOut
value: IqOutStruct = driver.route.scenario.iqOut.get()
```

No command help available

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

set(*tx_connector: TxConnector*, *tx_converter: TxConverter*) → None

```
# SCPI: ROUTe:GPRF:GENerator<Instance>:SCENario:IQOut
driver.route.scenario.iqOut.set(tx_connector = enums.TxConnector.I120, tx_
↪converter = enums.TxConverter.ITX1)
```

No command help available

param tx_connector
No help available

param tx_converter
No help available

6.2.1.2 Salone

SCPI Command :

```
ROUTe:GPRF:GENerator<Instance>:SCENario:SALone
```

class SaloneCls

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

class SaloneStruct

Response structure. Fields:

- Tx_Connector: enums.TxConnector: RF connector for the output path
- Rf_Converter: enums.TxConverter: TX module for the output path

get() → SaloneStruct

```
# SCPI: ROUTe:GPRF:GENerator<Instance>:SCENario:SALone
value: SaloneStruct = driver.route.scenario.salone.get()
```

Selects the output path for the generated RF signal. For possible connector and converter values, see 'Values for RF path selection'.

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

set(*tx_connector: TxConnector*, *rf_converter: TxConverter*) → None

```
# SCPI: ROUTe:GPRF:GENerator<Instance>:SCENario:SALone
driver.route.scenario.salone.set(tx_connector = enums.TxConnector.I120, rf_
↪converter = enums.TxConverter.ITX1)
```

Selects the output path for the generated RF signal. For possible connector and converter values, see 'Values for RF path selection'.

param tx_connector
RF connector for the output path

param rf_converter
TX module for the output path

6.3 Source

SCPI Command :

```
SOURce:GPRF:GENerator<Instance>:BBMode
```

class SourceCls

Source commands group definition. 168 total commands, 7 Subgroups, 1 group commands

get_bb_mode() → BasebandMode

```
# SCPI: SOURce:GPRF:GENerator<Instance>:BBMode
value: enums.BasebandMode = driver.source.get_bb_mode()
```

Selects the baseband mode for the generator signal.

return
baseband_mode: CW | DTONe | ARB CW: unmodulated CW signal DTONe: dual-tone signal ARB: ARB generator processing a waveform file

set_bb_mode(*baseband_mode: BasebandMode*) → None

```
# SCPI: SOURce:GPRF:GENerator<Instance>:BBMode
driver.source.set_bb_mode(baseband_mode = enums.BasebandMode.ARB)
```

Selects the baseband mode for the generator signal.

param baseband_mode
CW | DTONe | ARB CW: unmodulated CW signal DTONe: dual-tone signal ARB: ARB generator processing a waveform file

Cloning the Group

```
# Create a clone of the original group, that exists independently
group2 = driver.source.clone()
```

Subgroups

6.3.1 Arb

SCPI Commands :

```
SOURce:GPRF:GENerator<Instance>:ARB:FOFFset
SOURce:GPRF:GENerator<Instance>:ARB:SCOunt
SOURce:GPRF:GENerator<Instance>:ARB:ASAMples
SOURce:GPRF:GENerator<Instance>:ARB:REPetition
SOURce:GPRF:GENerator<Instance>:ARB:CYCLes
SOURce:GPRF:GENerator<Instance>:ARB:POFFset
SOURce:GPRF:GENerator<Instance>:ARB:CRATe
SOURce:GPRF:GENerator<Instance>:ARB:LOFFset
SOURce:GPRF:GENerator<Instance>:ARB:CRCProtect
SOURce:GPRF:GENerator<Instance>:ARB:STATus
```

class ArbCls

Arb commands group definition. 28 total commands, 6 Subgroups, 10 group commands

class ScountStruct

Structure for reading output parameters. Fields:

- Sample_Count_Time: float: No parameter help available
- Sample_Count: List[int]: No parameter help available

get_asamples() → int

```
# SCPI: SOURce:GPRF:GENerator<Instance>:ARB:ASAMples
value: int = driver.source.arb.get_asamples()
```

Extends the processing time of a waveform file by the specified number of samples. The additional samples are valid in single-shot repetition mode only, see method RsCmwGprfGen.Source.Arb.repetition.

return
add_samples: numeric Range: 0 to max. (depending on waveform file)

get_crate() → float

```
# SCPI: SOURce:GPRF:GENerator<Instance>:ARB:CRATe
value: float = driver.source.arb.get_crate()
```

Queries the clock rate of the loaded waveform file. Note: If a multi-segment waveform file is loaded, this command returns the clock rate of the last segment. Use method RsCmwGprfGen.Source.Arb.Msegment.crate to query the clock rates of the individual segments.

return
clock_rate: float Unit: Hz

get_crc_protect() → YesNoStatus

```
# SCPI: SOURce:GPRF:GENerator<Instance>:ARB:CRCProtect
value: enums.YesNoStatus = driver.source.arb.get_crc_protect()
```

Indicates whether the loaded ARB file contains a CRC checksum. To get a valid result, the related ARB file must be loaded into the memory. That means, the baseband mode must be ARB and the generator state must be ON. Otherwise, NAV is returned.

return
crc_protection: NO | YES

get_cycles() → int

```
# SCPI: SOURce:GPRF:GENerator<Instance>:ARB:CYCLes
value: int = driver.source.arb.get_cycles()
```

Defines how often the ARB file is processed. The ARB cycles are relevant in single-shot repetition mode only, see method RsCmwGprfGen.Source.Arb.repetition.

return
cycles: numeric Range: 1 to 10000

get_foffset() → float

```
# SCPI: SOURce:GPRF:GENerator<Instance>:ARB:FOFFset
value: float = driver.source.arb.get_foffset()
```

Sets the frequency offset to be imposed at the baseband during ARB generation.

return
frequency_offset: numeric Unit: Hz

get_loffset() → float

```
# SCPI: SOURce:GPRF:GENerator<Instance>:ARB:LOFFset
value: float = driver.source.arb.get_loffset()
```

Queries the level offset (peak to average ratio, PAR) of the loaded waveform file. The PAR is equal to the absolute value of the difference between the 'RMS Offset' and the 'Peak Offset' (crest factor) . Note: If a multi-segment waveform file is loaded, this command returns the PAR of the last segment. Use method RsCmwGprfGen.Source.Arb.Msegment.par to query the PAR values of the individual segments.

return
level_offset: float Unit: dB

get_poffset() → float

```
# SCPI: SOURce:GPRF:GENerator<Instance>:ARB:POFFset
value: float = driver.source.arb.get_poffset()
```

Queries the peak offset of the loaded waveform file. Note: If a multi-segment waveform file is loaded, this command returns the peak offset of the last segment. Use method RsCmwGprfGen.Source.Arb.Msegment.poffset to query the peak offset values of the individual segments.

return
peak_offset: float Unit: dB

get_repetition() → RepeatMode

```
# SCPI: SOURce:GPRF:GENerator<Instance>:ARB:REPetition
value: enums.RepeatMode = driver.source.arb.get_repetition()
```

Defines how often the ARB file is processed.

return
repetition: CONTinuous | SINGle CONTinuous: unlimited, cyclic processing SINGle: The file is processed n times, where n is the number of cycles, see method RsCmwGprfGen.Source.Arb.cycles.

get_scount() → ScountStruct

```
# SCPI: SOURce:GPRF:GENerator<Instance>:ARB:SCOunt
value: ScountStruct = driver.source.arb.get_scount()
```

Queries the progress of ARB file processing. You can use the command to check in single-shot mode whether ARB file processing is complete. As long as the ARB file is processed, the command returns 0,0,0. In continuous mode, the command always returns 0,0,0. If ARB file processing is complete, the command returns results for the previous ARB file processing.

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

get_status() → int

```
# SCPI: SOURce:GPRF:GENerator<Instance>:ARB:STATus
value: int = driver.source.arb.get_status()
```

Queries the number of the currently processed segment. Even for the repetition 'Continuous Seamless', the currently processed segment is returned, independent of whether a trigger event for the next segment has already been received or not.

return
arb_segment_no: decimal NAV is returned if no file is loaded.

set_asamples(*add_samples: int*) → None

```
# SCPI: SOURce:GPRF:GENerator<Instance>:ARB:ASAMples
driver.source.arb.set_asamples(add_samples = 1)
```

Extends the processing time of a waveform file by the specified number of samples. The additional samples are valid in single-shot repetition mode only, see method RsCmwGprfGen.Source.Arb.repetition.

param add_samples
numeric Range: 0 to max. (depending on waveform file)

set_cycles(*cycles: int*) → None

```
# SCPI: SOURce:GPRF:GENerator<Instance>:ARB:CYCLes
driver.source.arb.set_cycles(cycles = 1)
```

Defines how often the ARB file is processed. The ARB cycles are relevant in single-shot repetition mode only, see method RsCmwGprfGen.Source.Arb.repetition.

param cycles
numeric Range: 1 to 10000

set_foffset(*frequency_offset: float*) → None

```
# SCPI: SOURce:GPRF:GENerator<Instance>:ARB:FOFFset
driver.source.arb.set_foffset(frequency_offset = 1.0)
```

Sets the frequency offset to be imposed at the baseband during ARB generation.

param frequency_offset
numeric Unit: Hz

set_repetition(*repetition: RepeatMode*) → None

```
# SCPI: SOURce:GPRF:GENerator<Instance>:ARB:REPetition
driver.source.arb.set_repetition(repetition = enums.RepeatMode.CONTinuous)
```

Defines how often the ARB file is processed.

param repetition
CONTinuous | SINGle CONTinuous: unlimited, cyclic processing SINGle: The file is processed n times, where n is the number of cycles, see method RsCmwGprfGen.Source.Arb.cycles.

Cloning the Group

```
# Create a clone of the original group, that exists independently
group2 = driver.source.arb.clone()
```

Subgroups

6.3.1.1 File

SCPI Commands :

```
SOURce:GPRF:GENerator<Instance>:ARB:FILE
SOURce:GPRF:GENerator<Instance>:ARB:FILE:DATE
SOURce:GPRF:GENerator<Instance>:ARB:FILE:VERSion
SOURce:GPRF:GENerator<Instance>:ARB:FILE:OPTion
```

class FileCls

File commands group definition. 4 total commands, 0 Subgroups, 4 group commands

get(*arb_file: ArbFile = None*) → str

```
# SCPI: SOURce:GPRF:GENerator<Instance>:ARB:FILE
value: str = driver.source.arb.file.get(arb_file = enums.ArbFile.ABSPath)
```

Selects a waveform file for the ARB baseband mode. This command supports path aliases (e.g. @WAVEFORM) . Use MMEMory:ALIases? to query the available path aliases. If the selected file does not exist or no file has been selected, a query returns 'No File Selected'.

INTRO_CMD_HELP: If the selected file does exist, a query returns:

- Without <PathType>: The string used to select the file. If an alias has been used, the alias is not substituted.
- With <PathType>: The absolute path of the file. If an alias has been used, the alias is substituted.

param arb_file
(enum or string) string Name of the waveform file to be used (.wv) .

return
arb_file_retrun: No help available

get_date() → str

```
# SCPI: SOURce:GPRF:GENerator<Instance>:ARB:FILE:DATE
value: str = driver.source.arb.file.get_date()
```

Queries the date of the loaded waveform file.

return
date: string

get_option() → str

```
# SCPI: SOURce:GPRF:GENerator<Instance>:ARB:FILE:OPTion
value: str = driver.source.arb.file.get_option()
```

Returns the options that are required to play the loaded ARB file.

return
options: string A comma-separated list of options.

get_version() → str

```
# SCPI: SOURce:GPRF:GENerator<Instance>:ARB:FILE:VERSion
value: str = driver.source.arb.file.get_version()
```

Queries the version of the loaded waveform file.

return
version: string Version or empty string, if no file version is defined.

set(*arb_file: ArbFile = None*) → None

```
# SCPI: SOURce:GPRF:GENerator<Instance>:ARB:FILE
driver.source.arb.file.set(arb_file = enums.ArbFile.ABSPath)
```

Selects a waveform file for the ARB baseband mode. This command supports path aliases (e.g. @WAVEFORM) . Use MMEMory:ALIases? to query the available path aliases. If the selected file does not exist or no file has been selected, a query returns 'No File Selected'.

INTRO_CMD_HELP: If the selected file does exist, a query returns:

- Without <PathType>: The string used to select the file. If an alias has been used, the alias is not substituted.
- With <PathType>: The absolute path of the file. If an alias has been used, the alias is substituted.

param arb_file
(enum or string) string Name of the waveform file to be used (.wv) .

6.3.1.2 Marker

class MarkerCls

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

Cloning the Group

```
# Create a clone of the original group, that exists independently
group2 = driver.source.arb.marker.clone()
```

Subgroups

6.3.1.2.1 Delays

SCPI Command :

```
SOURce:GPRF:GENerator<Instance>:ARB:MARKer:DELays
```

class DelaysCls

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

class DelaysStruct

Response structure. Fields:

- Marker_2: int: numeric Range: -10 to 4000
- Marker_3: int: numeric Range: -10 to 4000
- Marker_4: int: numeric Range: -10 to 4000
- Restart_Marker: int: numeric Range: 0 to max. (depending on waveform file)

get() → DelaysStruct

```
# SCPI: SOURce:GPRF:GENerator<Instance>:ARB:MARKer:DELays
value: DelaysStruct = driver.source.arb.marker.delays.get()
```

Defines delay times for the ARB output trigger events relative to the marker events.

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

set(*marker_2: int*, *marker_3: int*, *marker_4: int*, *restart_marker: int*) → None

```
# SCPI: SOURce:GPRF:GENerator<Instance>:ARB:MARKer:DELays
driver.source.arb.marker.delays.set(marker_2 = 1, marker_3 = 1, marker_4 = 1,␣
↪restart_marker = 1)
```

Defines delay times for the ARB output trigger events relative to the marker events.

param marker_2
numeric Range: -10 to 4000

param marker_3
numeric Range: -10 to 4000

param marker_4
numeric Range: -10 to 4000

param restart_marker
numeric Range: 0 to max. (depending on waveform file)

6.3.1.3 Msegment

SCPI Commands :

```
SOURce:GPRF:GENerator<Instance>:ARB:MSEGment:NAME
SOURce:GPRF:GENerator<Instance>:ARB:MSEGment:POFFset
SOURce:GPRF:GENerator<Instance>:ARB:MSEGment:PAR
SOURce:GPRF:GENerator<Instance>:ARB:MSEGment:DURation
SOURce:GPRF:GENerator<Instance>:ARB:MSEGment:SAMPles
SOURce:GPRF:GENerator<Instance>:ARB:MSEGment:CRATe
SOURce:GPRF:GENerator<Instance>:ARB:MSEGment:NUMBer
```

class MsegmentCls

Msegment commands group definition. 7 total commands, 0 Subgroups, 7 group commands

get_crate() → List[float]

```
# SCPI: SOURce:GPRF:GENerator<Instance>:ARB:MSEGment:CRATe
value: List[float] = driver.source.arb.msegment.get_crate()
```

Queries the clock rates of all segments in the loaded multi-segment waveform file. The clock rates of waveform files created with R&S WinIQSIM2 are compatible with the R&S CMW.

return
clock_rate: float Comma-separated list of clock rates, one value per segment Unit: Hz

get_duration() → List[float]

```
# SCPI: SOURce:GPRF:GENerator<Instance>:ARB:MSEGment:DURation
value: List[float] = driver.source.arb.msegment.get_duration()
```

Queries the durations (processing times) of all segments in the loaded multi-segment waveform file. The duration is given by the number of samples divided by the clock rate.

return
duration: float Comma-separated list of durations, one value per segment Unit: s

get_name() → List[str]

```
# SCPI: SOURce:GPRF:GENerator<Instance>:ARB:MSEGment:NAME
value: List[str] = driver.source.arb.msegment.get_name()
```

Queries the names of all segments in the loaded multi-segment waveform file.

return
name: string Comma-separated list of names, one string per segment

get_number() → List[int]

```
# SCPI: SOURce:GPRF:GENerator<Instance>:ARB:MSEGment:NUMBer
value: List[int] = driver.source.arb.msegment.get_number()
```

Queries the segment numbers of all segments in the loaded multi-segment waveform file.

return
seg_number: decimal Comma-separated list of segment numbers, one value per segment

get_par() → List[float]

```
# SCPI: SOURce:GPRF:GENerator<Instance>:ARB:MSEGment:PAR
value: List[float] = driver.source.arb.msegment.get_par()
```

Queries the level offset (peak to average ratio, PAR) of all segments in the loaded multi-segment waveform file. The PAR is equal to the absolute value of the difference between the 'RMS Offset' and the 'Peak Offset' defined in WinIQSIM2 (crest factor) .

return
par: float Comma-separated list of PAR values, one value per segment Unit: dB

get_poffset() → List[float]

```
# SCPI: SOURce:GPRF:GENerator<Instance>:ARB:MSEGment:POFFset
value: List[float] = driver.source.arb.msegment.get_poffset()
```

Queries the peak offset of all segments in the loaded multi-segment waveform file.

return
peak_offset: float Comma-separated list of peak offset values, one value per segment Unit: dB

get_samples() → List[int]

```
# SCPI: SOURce:GPRF:GENerator<Instance>:ARB:MSEGment:SAMPles
value: List[int] = driver.source.arb.msegment.get_samples()
```

Queries the number of samples in all segments in the loaded multi-segment waveform file.

return
samples: decimal Comma-separated list of sample values, one value per segment

6.3.1.4 Samples

SCPI Command :

```
SOURce:GPRF:GENerator<Instance>:ARB:SAMPles
```

class SamplesCls

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

get_value() → float

```
# SCPI: SOURce:GPRF:GENerator<Instance>:ARB:SAMPles
value: float = driver.source.arb.samples.get_value()
```

Queries the number of samples in the loaded waveform file. The R&S CMW supports waveform files with a size up to 512 MB.

return
samples: float

Cloning the Group

```
# Create a clone of the original group, that exists independently
group2 = driver.source.arb.samples.clone()
```

Subgroups

6.3.1.4.1 Range

SCPI Command :

```
SOURce:GPRF:GENerator<Instance>:ARB:SAMPles:RANGe
```

class RangeCls

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

class RangeStruct

Response structure. Fields:

- Range_Py: enums.Range: FULL | SUB FULL: The full ARB file is processed. SUB: The subrange defined by the Start and Stop parameters is processed.
- Start: int: integer The beginning (first sample) of the subrange. Range: 0 (fixed)
- Stop: int: integer The end (last sample) of the subrange. Range: 16 to #samples in loaded ARB file - 1

get() → RangeStruct

```
# SCPI: SOURce:GPRF:GENerator<Instance>:ARB:SAMPles:RANGe
value: RangeStruct = driver.source.arb.samples.range.get()
```

Defines the range of samples in the loaded ARB file that are processed.

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

set(*range_py: Range*, *start: int = None*, *stop: int = None*) → None

```
# SCPI: SOURce:GPRF:GENerator<Instance>:ARB:SAMPles:RANGe
driver.source.arb.samples.range.set(range_py = enums.Range.FULL, start = 1,␣
↪stop = 1)
```

Defines the range of samples in the loaded ARB file that are processed.

param range_py
FULL | SUB FULL: The full ARB file is processed. SUB: The subrange defined by the Start and Stop parameters is processed.

param start
integer The beginning (first sample) of the subrange. Range: 0 (fixed)

param stop
integer The end (last sample) of the subrange. Range: 16 to #samples in loaded ARB file - 1

6.3.1.5 Segments

SCPI Commands :

```
SOURce:GPRF:GENerator<Instance>:ARB:SEGMents:NEXT
SOURce:GPRF:GENerator<Instance>:ARB:SEGMents:CURRent
```

class SegmentsCls

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

class CurrentStruct

Structure for reading output parameters. Fields:

- Segment_Number: int: decimal NAV is returned if no file is loaded.
- Segment_Name: str: string NAV is returned if no file is loaded or no name is defined.

get_current() → CurrentStruct

```
# SCPI: SOURce:GPRF:GENerator<Instance>:ARB:SEGMents:CURRent
value: CurrentStruct = driver.source.arb.segments.get_current()
```

Queries the number and name of the currently processed segment. For the repetition 'Continuous Seamless', a trigger event has been received for the returned segment. The generator is still processing the previous segment or it is already processing the returned segment. For a distinction of the two cases, see method RsCmwGprfGen.Source.Arb.status.

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

get_next() → int

```
# SCPI: SOURce:GPRF:GENerator<Instance>:ARB:SEGMents:NEXT
value: int = driver.source.arb.segments.get_next()
```

Selects a segment to be processed after the end of the currently processed segment.

return
segment_number: numeric

set_next(*segment_number: int*) → None

```
# SCPI: SOURce:GPRF:GENerator<Instance>:ARB:SEGMents:NEXT
driver.source.arb.segments.set_next(segment_number = 1)
```

Selects a segment to be processed after the end of the currently processed segment.

param segment_number
numeric

6.3.1.6 UdMarker

SCPI Command :

```
SOURce:GPRF:GENerator<Instance>:ARB:UDMarker
```

class UdMarkerCls

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

class UdMarkerStruct

Response structure. Fields:

- Period: int: No parameter help available
- Start_State: enums.SignalSlope: No parameter help available
- Positions: List[int or bool]: No parameter help available

get() → UdMarkerStruct

```
# SCPI: SOURce:GPRF:GENerator<Instance>:ARB:UDMarker
value: UdMarkerStruct = driver.source.arb.udMarker.get()
```

No command help available

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

set(*period: int*, *start_state: SignalSlope*, *positions: List[int]*) → None

```
# SCPI: SOURce:GPRF:GENerator<Instance>:ARB:UDMarker
driver.source.arb.udMarker.set(period = 1, start_state = enums.SignalSlope.
↪FEDGe, positions = [1, True, 2, False, 3])
```

No command help available

param period
No help available

param start_state
No help available

param positions
(integer or boolean items) No help available

Cloning the Group

```
# Create a clone of the original group, that exists independently
group2 = driver.source.arb.udMarker.clone()
```

Subgroups

6.3.1.6.1 Clist

SCPI Command :

```
SOURce:GPRF:GENerator<Instance>:ARB:UDMarker:CLISt
```

class ClistCls

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

set() → None

```
# SCPI: SOURce:GPRF:GENerator<Instance>:ARB:UDMarker:CLISt
driver.source.arb.udMarker.clist.set()
```

No command help available

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

```
# SCPI: SOURce:GPRF:GENerator<Instance>:ARB:UDMarker:CLISt
driver.source.arb.udMarker.clist.set_with_opc()
```

No command help available

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

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

6.3.2 Dtone

SCPI Command :

```
SOURce:GPRF:GENerator<Instance>:DTONe:RATio
```

class DtoneCls

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

get_ratio() → float

```
# SCPI: SOURce:GPRF:GENerator<Instance>:DTONe:RATio
value: float = driver.source.dtone.get_ratio()
```

Specifies the ratio in dB between the RMS levels of the two signals.

return
ratio: numeric Unit: dB

set_ratio(*ratio: float*) → None

```
# SCPI: SOURce:GPRF:GENerator<Instance>:DTONe:RATio
driver.source.dtone.set_ratio(ratio = 1.0)
```

Specifies the ratio in dB between the RMS levels of the two signals.

param ratio
numeric Unit: dB

Cloning the Group

```
# Create a clone of the original group, that exists independently
group2 = driver.source.dtone.clone()
```

Subgroups

6.3.2.1 Level<LevelSource>

RepCap Settings

```
# Range: Src1 .. Src2
rc = driver.source.dtone.level.repcap_levelSource_get()
driver.source.dtone.level.repcap_levelSource_set(repcap.LevelSource.Src1)
```

SCPI Command :

```
SOURce:GPRF:GENerator<Instance>:DTONe:LEVel<source>
```

class LevelCls

Level commands group definition. 1 total commands, 0 Subgroups, 1 group commands Repeated Capability: LevelSource, default value after init: LevelSource.Src1

get(*levelSource=LevelSource.Default*) → float

```
# SCPI: SOURce:GPRF:GENerator<Instance>:DTONe:LEVel<source>
value: float = driver.source.dtone.level.get(levelSource = repcap.LevelSource.
↪Default)
```

Queries the output level of a source signal. The output level is a function of the generator output level and the ratio, see method RsCmwGprfGen.Source.RfSettings.level and method RsCmwGprfGen.Source.Dtone.ratio.

param levelSource
optional repeated capability selector. Default value: Src1 (settable in the interface 'Level')

return
level: float Range: Please notice the ranges quoted in the data sheet. , Unit: dBm

Cloning the Group

```
# Create a clone of the original group, that exists independently
group2 = driver.source.dtone.level.clone()
```

6.3.2.2 Ofrequency<FrequencySource>

RepCap Settings

```
# Range: Src1 .. Src2
rc = driver.source.dtone.ofrequency.repcap_frequencySource_get()
driver.source.dtone.ofrequency.repcap_frequencySource_set(repcap.FrequencySource.Src1)
```

SCPI Command :

```
SOURce:GPRF:GENerator<Instance>:DTONe:OFRequency<source>
```

class OfrequencyCls

Ofrequency commands group definition. 1 total commands, 0 Subgroups, 1 group commands Repeated Capability: FrequencySource, default value after init: FrequencySource.Src1

get(*frequencySource=FrequencySource.Default*) → float

```
# SCPI: SOURce:GPRF:GENerator<Instance>:DTONe:OFRequency<source>
value: float = driver.source.dtone.ofrequency.get(frequencySource = repcap.
↪FrequencySource.Default)
```

Selects an offset frequency. The frequency of the modulated signal is equal to the base frequency (see method RsCmwGprfGen.Source.RfSettings.frequency) plus the offset frequency.

param frequencySource
optional repeated capability selector. Default value: Src1 (settable in the interface 'Ofrequency')

return
frequency: numeric Unit: Hz

set(*frequency: float*, *frequencySource=FrequencySource.Default*) → None

```
# SCPI: SOURce:GPRF:GENerator<Instance>:DTONe:OFRequency<source>
driver.source.dtone.ofrequency.set(frequency = 1.0, frequencySource = repcap.
↪FrequencySource.Default)
```

Selects an offset frequency. The frequency of the modulated signal is equal to the base frequency (see method RsCmwGprfGen.Source.RfSettings.frequency) plus the offset frequency.

param frequency
numeric Unit: Hz

param frequencySource
optional repeated capability selector. Default value: Src1 (settable in the interface 'Ofrequency')

Cloning the Group

```
# Create a clone of the original group, that exists independently
group2 = driver.source.dtone.ofrequency.clone()
```

6.3.3 IqSettings

SCPI Commands :

```
SOURce:GPRF:GENerator<Instance>:IQSettings:SRATe
SOURce:GPRF:GENerator<Instance>:IQSettings:TMODe
SOURce:GPRF:GENerator<Instance>:IQSettings:LEVel
SOURce:GPRF:GENerator<Instance>:IQSettings:PEP
SOURce:GPRF:GENerator<Instance>:IQSettings:CRESt
```

class IqSettingsCls

IqSettings commands group definition. 5 total commands, 0 Subgroups, 5 group commands

get_crest() → float

```
# SCPI: SOURce:GPRF:GENerator<Instance>:IQSettings:CRESt
value: float = driver.source.iqSettings.get_crest()
```

No command help available

return
crest: No help available

get_level() → float

```
# SCPI: SOURce:GPRF:GENerator<Instance>:IQSettings:LEVel
value: float = driver.source.iqSettings.get_level()
```

No command help available

return
level: No help available

get_pep() → float

```
# SCPI: SOURce:GPRF:GENerator<Instance>:IQSettings:PEP
value: float = driver.source.iqSettings.get_pep()
```

No command help available

return
pep: No help available

get_symbol_rate() → float

```
# SCPI: SOURce:GPRF:GENerator<Instance>:IQSettings:SRATe
value: float = driver.source.iqSettings.get_symbol_rate()
```

No command help available

return
sample_rate: No help available

get_tmode() → TransferMode

```
# SCPI: SOURce:GPRF:GENerator<Instance>:IQSettings:TMODe
value: enums.TransferMode = driver.source.iqSettings.get_tmode()
```

No command help available

return
transfer_mode: No help available

set_symbol_rate(*sample_rate: float*) → None

```
# SCPI: SOURce:GPRF:GENerator<Instance>:IQSettings:SRATe
driver.source.iqSettings.set_symbol_rate(sample_rate = 1.0)
```

No command help available

param sample_rate
No help available

set_tmode(*transfer_mode: TransferMode*) → None

```
# SCPI: SOURce:GPRF:GENerator<Instance>:IQSettings:TMODe
driver.source.iqSettings.set_tmode(transfer_mode = enums.TransferMode.
↪ENABlemode)
```

No command help available

param transfer_mode
No help available

6.3.4 ListPy

SCPI Commands :

```
SOURce:GPRF:GENerator<Instance>:LIST:AINDex
SOURce:GPRF:GENerator<Instance>:LIST:FILL
SOURce:GPRF:GENerator<Instance>:LIST:GOTO
SOURce:GPRF:GENerator<Instance>:LIST:REPetition
SOURce:GPRF:GENerator<Instance>:LIST:STARt
SOURce:GPRF:GENerator<Instance>:LIST:STOP
SOURce:GPRF:GENerator<Instance>:LIST:COUNt
SOURce:GPRF:GENerator<Instance>:LIST
```

class ListPyCls

ListPy commands group definition. 32 total commands, 13 Subgroups, 8 group commands

class FillStruct

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

- Start_Index: float: numeric The start index of the list segment to be ‘filled’. Range: 0 to 1999
- Range_Py: float: numeric The range (length) of the list segment to be ‘filled’. Range: 1 to 2000

- Index_Repetition: int: integer The constant 'Index Repetition' within this list segment. Range: 1 to 10000
- Start_Frequency: float: numeric The frequency of list item StartIndex. For the supported frequency range, see 'Frequency ranges'. Unit: Hz
- Freq_Increment: float: numeric The frequency increment within this list segment. Range: -282.45 MHz to 1.20005 GHz , Unit: Hz
- Start_Power: float: numeric The RMS level of list item StartIndex. Range: Depends on the instrument model, the connector and other settings; please notice the ranges quoted in the data sheet , Unit: dBm
- Power_Increment: float: numeric The power increment within this list segment. Range: -29.5 dBm to 3 dBm , Unit: dBm
- Start_Dwell_Time: float: Optional setting parameter. numeric The constant dwell time within this list segment. Range: 2.0E-4 s to 20 s , Unit: s
- Reenable: bool: Optional setting parameter. OFF | ON The constant 'Reenable' property within this list segment.
- Modulation: bool: Optional setting parameter. OFF | ON The constant 'Modulation ON|OFF' property within this list segment.
- Start_Gain: float: Optional setting parameter. numeric The digital gain of list item StartIndex. Range: -30 dB to 0 dB , Unit: dB
- Gain_Increment: float: Optional setting parameter. numeric The digital gain increment within this list segment. Range: -7.5 dB to 0 dB , Unit: dB

get_aindex() → int

```
# SCPI: SOURce:GPRF:GENerator<Instance>:LIST:AINDex
value: int = driver.source.listPy.get_aindex()
```

Returns the currently active list index.

return
active_index: decimal Range: 0 to 19

get_count() → int

```
# SCPI: SOURce:GPRF:GENerator<Instance>:LIST:COUNt
value: int = driver.source.listPy.get_count()
```

Queries the number of frequency/level steps of the RF generator in list mode.

return
list_count: decimal Number of frequency/level steps in list mode Range: 1 to 2000

get_goto() → int

```
# SCPI: SOURce:GPRF:GENerator<Instance>:LIST:GOTO
value: int = driver.source.listPy.get_goto()
```

Defines the start index for the second and all following generator cycles in continuous mode (method RsCmwGprfGen.Source. ListPy.repetition) . The index must be in the selected list section (method RsCmwGprfGen.Source.ListPy.Sstop.set) .

return
goto_index: numeric Range: 1 to 2000

get_repetition() → RepeatMode

```
# SCPI: SOURce:GPRF:GENerator<Instance>:LIST:REPetition
value: enums.RepeatMode = driver.source.listPy.get_repetition()
```

Defines how often the RF generator runs through the list.

return
repetition: CONTinuous | SINGle CONTinuous: The generator cycles through the list. SINGle: The generator runs through the list for a single time. The sequence is triggered via method RsCmwGprfGen.Source.ListPy.Esingle.set.

get_start() → int

```
# SCPI: SOURce:GPRF:GENerator<Instance>:LIST:STARt
value: int = driver.source.listPy.get_start()
```

Defines the number of the first measured frequency/level step in the list. The start index must not be larger than the stop index (see method RsCmwGprfGen.Source.ListPy.stop) .

return
start_index: numeric Range: 0 to 1999

get_stop() → int

```
# SCPI: SOURce:GPRF:GENerator<Instance>:LIST:STOP
value: int = driver.source.listPy.get_stop()
```

Defines the number of the last measured frequency/level step in the list. The stop index must not be smaller than the start index (see method RsCmwGprfGen.Source.ListPy.start) .

return
stop_index: numeric Range: 0 to 1999

get_value() → bool

```
# SCPI: SOURce:GPRF:GENerator<Instance>:LIST
value: bool = driver.source.listPy.get_value()
```

Enables or disables the list mode of the RF generator.

return
enable_list_mode: ON | OFF ON: List mode enabled OFF: List mode disabled (constant-frequency generator)

set_fill(*value: FillStruct*) → None

```
# SCPI: SOURce:GPRF:GENerator<Instance>:LIST:FILL
structure = driver.source.listPy.FillStruct()
structure.Start_Index: float = 1.0
structure.Range_Py: float = 1.0
structure.Index_Repetition: int = 1
structure.Start_Frequency: float = 1.0
structure.Freq_Increment: float = 1.0
structure.Start_Power: float = 1.0
structure.Power_Increment: float = 1.0
structure.Start_Dwell_Time: float = 1.0
```

(continues on next page)

(continued from previous page)

```
structure.Reenable: bool = False
structure.Modulation: bool = False
structure.Start_Gain: float = 1.0
structure.Gain_Increment: float = 1.0
driver.source.listPy.set_fill(value = structure)
```

Convenience command to simplify the configuration of the frequency/level list. Within a list segment determined by its start index and range (length) , the frequency, power and (optionally) the digital gain are incremented by configurable step sizes. The other list item settings are fixed.

param value
see the help for FillStruct structure arguments.

set_goto(*goto_index: int*) → None

```
# SCPI: SOURce:GPRF:GENerator<Instance>:LIST:GOTO
driver.source.listPy.set_goto(goto_index = 1)
```

Defines the start index for the second and all following generator cycles in continuous mode (method RsCmwGprfGen.Source. ListPy.repetition) . The index must be in the selected list section (method RsCmwGprfGen.Source.ListPy.Sstop.set) .

param goto_index
numeric Range: 1 to 2000

set_repetition(*repetition: RepeatMode*) → None

```
# SCPI: SOURce:GPRF:GENerator<Instance>:LIST:REPetition
driver.source.listPy.set_repetition(repetition = enums.RepeatMode.CONTinuous)
```

Defines how often the RF generator runs through the list.

param repetition
CONTinuous | SINGle CONTinuous: The generator cycles through the list. SINGle: The generator runs through the list for a single time. The sequence is triggered via method RsCmwGprfGen.Source.ListPy.Esingle.set.

set_start(*start_index: int*) → None

```
# SCPI: SOURce:GPRF:GENerator<Instance>:LIST:STARt
driver.source.listPy.set_start(start_index = 1)
```

Defines the number of the first measured frequency/level step in the list. The start index must not be larger than the stop index (see method RsCmwGprfGen.Source.ListPy.stop) .

param start_index
numeric Range: 0 to 1999

set_stop(*stop_index: int*) → None

```
# SCPI: SOURce:GPRF:GENerator<Instance>:LIST:STOP
driver.source.listPy.set_stop(stop_index = 1)
```

Defines the number of the last measured frequency/level step in the list. The stop index must not be smaller than the start index (see method RsCmwGprfGen.Source.ListPy.start) .

param stop_index
numeric Range: 0 to 1999

`set_value`(*enable_list_mode: bool*) → None

```
# SCPI: SOURce:GPRF:GENerator<Instance>:LIST
driver.source.listPy.set_value(enable_list_mode = False)
```

Enables or disables the list mode of the RF generator.

param enable_list_mode
ON | OFF ON: List mode enabled OFF: List mode disabled (constant-frequency generator)

Cloning the Group

```
# Create a clone of the original group, that exists independently
group2 = driver.source.listPy.clone()
```

Subgroups

6.3.4.1 Dgain

SCPI Commands :

```
SOURce:GPRF:GENerator<Instance>:LIST:DGAin
SOURce:GPRF:GENerator<Instance>:LIST:DGAin:ALL
```

`class DgainCls`

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

`get`(*index: int*) → float

```
# SCPI: SOURce:GPRF:GENerator<Instance>:LIST:DGAin
value: float = driver.source.listPy.dgain.get(index = 1)
```

Defines the digital gain of a selected frequency/level step.

param index
integer Number of the frequency/level step in the table Range: 0 to 1999

return
digital_gain: numeric Digital gain at the step Unit: dB

`get_all`() → List[float]

```
# SCPI: SOURce:GPRF:GENerator<Instance>:LIST:DGAin:ALL
value: List[float] = driver.source.listPy.dgain.get_all()
```

Defines the digital gains of all frequency/level steps.

return
all_digital_gains: numeric Comma-separated list of n values, one per frequency/level step, where n 2001. The query returns 2000 results. Unit: dB

set(*index: int*, *digital_gain: float*) → None

```
# SCPI: SOURce:GPRF:GENerator<Instance>:LIST:DGAin
driver.source.listPy.dgain.set(index = 1, digital_gain = 1.0)
```

Defines the digital gain of a selected frequency/level step.

param index
integer Number of the frequency/level step in the table Range: 0 to 1999

param digital_gain
numeric Digital gain at the step Unit: dB

set_all(*all_digital_gains: List[float]*) → None

```
# SCPI: SOURce:GPRF:GENerator<Instance>:LIST:DGAin:ALL
driver.source.listPy.dgain.set_all(all_digital_gains = [1.1, 2.2, 3.3])
```

Defines the digital gains of all frequency/level steps.

param all_digital_gains
numeric Comma-separated list of n values, one per frequency/level step, where n 2001. The query returns 2000 results. Unit: dB

6.3.4.2 Dtime

SCPI Commands :

```
SOURce:GPRF:GENerator<Instance>:LIST:DTIMe
SOURce:GPRF:GENerator<Instance>:LIST:DTIMe:ALL
```

class DtimeCls

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

get(*index: int*) → float

```
# SCPI: SOURce:GPRF:GENerator<Instance>:LIST:DTIMe
value: float = driver.source.listPy.dtime.get(index = 1)
```

Defines or queries the transmission time for a selected frequency/level step in 'Dwell Time' mode. The value is not used in the other list modes (see method RsCmwGprfGen.Source.ListPy.Increment.value) .

param index
integer Number of the frequency/level step in the table. Range: 0 to 1999

return
dwell_time: numeric Dwell time for the frequency/level step. Range: 200E-6 s to 10 s, Unit: s

get_all() → List[float]

```
# SCPI: SOURce:GPRF:GENerator<Instance>:LIST:DTIMe:ALL
value: List[float] = driver.source.listPy.dtime.get_all()
```

Defines the transmission times for all frequency/level steps in 'Dwell Time' mode. The value is not used in the other list modes (see method RsCmwGprfGen.Source.ListPy.Increment.value) .

return
all_dwelltimes: numeric Comma-separated list of n values, one per frequency/level step, where n 2001. The query returns 2000 results. Range: 200E-6 s to 10 s, Unit: s

set(*index: int*, *dwell_time: float*) → None

```
# SCPI: SOURce:GPRF:GENerator<Instance>:LIST:DTIMe
driver.source.listPy.dtime.set(index = 1, dwell_time = 1.0)
```

Defines or queries the transmission time for a selected frequency/level step in 'Dwell Time' mode. The value is not used in the other list modes (see method RsCmwGprfGen.Source.ListPy.Increment.value) .

param index
integer Number of the frequency/level step in the table. Range: 0 to 1999

param dwell_time
numeric Dwell time for the frequency/level step. Range: 200E-6 s to 10 s, Unit: s

set_all(*all_dwelltimes: List[float]*) → None

```
# SCPI: SOURce:GPRF:GENerator<Instance>:LIST:DTIMe:ALL
driver.source.listPy.dtime.set_all(all_dwelltimes = [1.1, 2.2, 3.3])
```

Defines the transmission times for all frequency/level steps in 'Dwell Time' mode. The value is not used in the other list modes (see method RsCmwGprfGen.Source.ListPy.Increment.value) .

param all_dwelltimes
numeric Comma-separated list of n values, one per frequency/level step, where n 2001. The query returns 2000 results. Range: 200E-6 s to 10 s, Unit: s

6.3.4.3 Esingle

SCPI Command :

```
SOURce:GPRF:GENerator<Instance>:LIST:ESINgle
```

class EsingleCls

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

set() → None

```
# SCPI: SOURce:GPRF:GENerator<Instance>:LIST:ESINgle
driver.source.listPy.esingle.set()
```

Starts a single generator cycle through the frequency/level list.
INTRO_CMD_HELP: This command is available only if:

- The list mode is enabled (see method RsCmwGprfGen.Source.ListPy.value) .
- And the 'Single' list mode is set (method RsCmwGprfGen.Source.ListPy.repetition) .
- And the increment 'Dwell Time' is set (method RsCmwGprfGen.Source.ListPy.Increment.value) .

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

```
# SCPI: SOURce:GPRF:GENerator<Instance>:LIST:ESINgle
driver.source.listPy.esingle.set_with_opc()
```

Starts a single generator cycle through the frequency/level list.
INTRO_CMD_HELP: This command is available only if:

- The list mode is enabled (see method RsCmwGprfGen.Source.ListPy.value) .
- And the 'Single' list mode is set (method RsCmwGprfGen.Source.ListPy.repetition) .
- And the increment 'Dwell Time' is set (method RsCmwGprfGen.Source.ListPy.Increment.value) .

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

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

6.3.4.4 Frequency

SCPI Commands :

```
SOURce:GPRF:GENerator<Instance>:LIST:FREQuency
SOURce:GPRF:GENerator<Instance>:LIST:FREQuency:ALL
```

class FrequencyCls

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

get(*index: int*) → float

```
# SCPI: SOURce:GPRF:GENerator<Instance>:LIST:FREQuency
value: float = driver.source.listPy.frequency.get(index = 1)
```

Defines or queries the frequency of a selected frequency/level step. For the supported frequency range, see 'Frequency ranges'.

param index
integer Number of the frequency/level step in the table Range: 0 to 1999

return
frequency: numeric Frequency of the frequency/level step Unit: Hz

get_all() → List[float]

```
# SCPI: SOURce:GPRF:GENerator<Instance>:LIST:FREQuency:ALL
value: List[float] = driver.source.listPy.frequency.get_all()
```

Defines the frequencies of all frequency/level steps. For the supported frequency range, see 'Frequency ranges'.

return
all_frequencies: numeric Comma-separated list of n values, one per frequency/level step, where n 2001. The query returns 2000 results. Unit: Hz

set(*index: int*, *frequency: float*) → None

```
# SCPI: SOURce:GPRF:GENerator<Instance>:LIST:FREQuency
driver.source.listPy.frequency.set(index = 1, frequency = 1.0)
```

Defines or queries the frequency of a selected frequency/level step. For the supported frequency range, see 'Frequency ranges'.

param index
integer Number of the frequency/level step in the table Range: 0 to 1999

param frequency
numeric Frequency of the frequency/level step Unit: Hz

set_all(*all_frequencies: List[float]*) → None

```
# SCPI: SOURce:GPRF:GENerator<Instance>:LIST:FREQuency:ALL
driver.source.listPy.frequency.set_all(all_frequencies = [1.1, 2.2, 3.3])
```

Defines the frequencies of all frequency/level steps. For the supported frequency range, see 'Frequency ranges'.

param all_frequencies
numeric Comma-separated list of n values, one per frequency/level step, where n 2001. The query returns 2000 results. Unit: Hz

6.3.4.5 Increment

SCPI Commands :

```
SOURce:GPRF:GENerator<Instance>:LIST:INCRement:CATalog
SOURce:GPRF:GENerator<Instance>:LIST:INCRement
```

class IncrementCls

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

get_catalog() → List[str]

```
# SCPI: SOURce:GPRF:GENerator<Instance>:LIST:INCRement:CATalog
value: List[str] = driver.source.listPy.increment.get_catalog()
```

Lists all list increment modes that can be set using method RsCmwGprfGen.Source.ListPy.Increment.value.

return
list_incr_srcs: string Comma-separated list of strings. Each string represents a supported increment mode.

get_value() → str

```
# SCPI: SOURce:GPRF:GENerator<Instance>:LIST:INCRement
value: str = driver.source.listPy.increment.get_value()
```

Defines how the RF generator steps through the list.

return
list_incr_src: string Source for the list increment. Examples: 'Dwell Time' The generator transmits at each frequency/level step for the selected dwell time (method

RsCmwGprfGen.Source.ListPy.Dtime.set) . 'GPRF Geni: some marker' List incremented by a marker in the played-back ARB file (only for baseband mode ARB, see method RsCmwGprfGen.Source.bbMode) . 'Some measurement' (e.g. 'GPRF Measi: Power') List incremented in line with a running measurement. Use method RsCmwGprfGen.Source.ListPy.Increment.catalog to query the list of possible sources for the current HW/SW configuration.

set_value(*list_incr_src: str*) → None

```
# SCPI: SOURce:GPRF:GENerator<Instance>:LIST:INCRement
driver.source.listPy.increment.set_value(list_incr_src = 'abc')
```

Defines how the RF generator steps through the list.

param list_incr_src
string Source for the list increment. Examples: 'Dwell Time' The generator transmits at each frequency/level step for the selected dwell time (method RsCmwGprfGen.Source.ListPy.Dtime.set) . 'GPRF Geni: some marker' List incremented by a marker in the played-back ARB file (only for baseband mode ARB, see method RsCmwGprfGen.Source.bbMode) . 'Some measurement' (e.g. 'GPRF Measi: Power') List incremented in line with a running measurement. Use method RsCmwGprfGen.Source.ListPy.Increment.catalog to query the list of possible sources for the current HW/SW configuration.

Cloning the Group

```
# Create a clone of the original group, that exists independently
group2 = driver.source.listPy.increment.clone()
```

Subgroups

6.3.4.5.1 Enabling

SCPI Commands :

```
SOURce:GPRF:GENerator<Instance>:LIST:INCRement:ENABling:CATalog
SOURce:GPRF:GENerator<Instance>:LIST:INCRement:ENABling
```

class EnablingCls

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

get_catalog() → List[str]

```
# SCPI: SOURce:GPRF:GENerator<Instance>:LIST:INCRement:ENABling:CATalog
value: List[str] = driver.source.listPy.increment.enabling.get_catalog()
```

Lists all initial trigger modes that can be set using method RsCmwGprfGen.Source.ListPy.Increment.Enabling.value.

return
enabling_srcs: string Comma-separated list of strings. Each string represents a supported initial trigger mode.

get_value() → str

```
# SCPI: SOURce:GPRF:GENerator<Instance>:LIST:INCRement:ENABling
value: str = driver.source.listPy.increment.enabling.get_value()
```

For an internally incremented list, this command defines the initial trigger. Internally incremented list means 'List Increment: GPRF Gen<i>: ... Marker ...' or 'List Increment: Dwell Time', see method RsCmwGprfGen.Source.ListPy.Increment. value.

return
enabling: string To generate a complete list of all supported triggers, see method RsCmwGprfGen.Source.ListPy.Increment.Enabling.catalog. 'Immediate': No initial trigger; list increment starts immediately. 'Manual': Waits until method RsCmwGprfGen.Source.ListPy.Slist.set is executed. 'meas trigger' (e.g. 'GPRF Measi: Power' or 'GSM Measi: Multi Evaluation') : Some measurement application provides the initial trigger.

set_value(*enabling: str*) → None

```
# SCPI: SOURce:GPRF:GENerator<Instance>:LIST:INCRement:ENABling
driver.source.listPy.increment.enabling.set_value(enabling = 'abc')
```

For an internally incremented list, this command defines the initial trigger. Internally incremented list means 'List Increment: GPRF Gen<i>: ... Marker ...' or 'List Increment: Dwell Time', see method RsCmwGprfGen.Source.ListPy.Increment. value.

param enabling
string To generate a complete list of all supported triggers, see method RsCmwGprfGen.Source.ListPy.Increment.Enabling.catalog. 'Immediate': No initial trigger; list increment starts immediately. 'Manual': Waits until method RsCmwGprfGen.Source.ListPy.Slist.set is executed. 'meas trigger' (e.g. 'GPRF Measi: Power' or 'GSM Measi: Multi Evaluation') : Some measurement application provides the initial trigger.

6.3.4.6 Irepetition

SCPI Commands :

```
SOURce:GPRF:GENerator<Instance>:LIST:IREPetition
SOURce:GPRF:GENerator<Instance>:LIST:IREPetition:ALL
```

class IrepetitionCls

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

get(*index: int*) → int

```
# SCPI: SOURce:GPRF:GENerator<Instance>:LIST:IREPetition
value: int = driver.source.listPy.irepetition.get(index = 1)
```

Defines or queries the 'Index Repetition' of a selected frequency/level step.

param index
integer Number of the frequency/level step in the table Range: 0 to 1999

return
repetition: numeric Repetition of the frequency/level step Range: 1 to 10E+3

get_all() → List[int]

```
# SCPI: SOURce:GPRF:GENerator<Instance>:LIST:IREPetition:ALL
value: List[int] = driver.source.listPy.irepetition.get_all()
```

Defines or queries the 'Index Repetition' of all frequency/level steps.

return
index_repetitions: numeric Comma-separated list of n values, one value per frequency/level step, where n 2000. The query returns 2000 results. Range: 1 to 10000

set(*index: int*, *repetition: int*) → None

```
# SCPI: SOURce:GPRF:GENerator<Instance>:LIST:IREPetition
driver.source.listPy.irepetition.set(index = 1, repetition = 1)
```

Defines or queries the 'Index Repetition' of a selected frequency/level step.

param index
integer Number of the frequency/level step in the table Range: 0 to 1999

param repetition
numeric Repetition of the frequency/level step Range: 1 to 10E+3

set_all(*index_repetitions: List[int]*) → None

```
# SCPI: SOURce:GPRF:GENerator<Instance>:LIST:IREPetition:ALL
driver.source.listPy.irepetition.set_all(index_repetitions = [1, 2, 3])
```

Defines or queries the 'Index Repetition' of all frequency/level steps.

param index_repetitions
numeric Comma-separated list of n values, one value per frequency/level step, where n 2000. The query returns 2000 results. Range: 1 to 10000

6.3.4.7 Modulation

SCPI Commands :

```
SOURce:GPRF:GENerator<Instance>:LIST:MODulation
SOURce:GPRF:GENerator<Instance>:LIST:MODulation:ALL
```

class ModulationCls

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

get(*index: int*) → bool

```
# SCPI: SOURce:GPRF:GENerator<Instance>:LIST:MODulation
value: bool = driver.source.listPy.modulation.get(index = 1)
```

Defines or queries the 'Mod. On / Off' setting of a selected frequency/level step. The setting is valid only in arbitrary baseband mode (see method RsCmwGprfGen.Source.bbMode) .

param index
integer Number of the frequency/level step in the table Range: 0 to 1999

return
modulation: OFF | ON Switch modulation OFF or ON

`get_all`() → List[bool]

```
# SCPI: SOURce:GPRF:GENerator<Instance>:LIST:MODulation:ALL
value: List[bool] = driver.source.listPy.modulation.get_all()
```

Defines or queries the 'Mod. On / Off' setting of all frequency/level steps. The setting is valid only in arbitrary baseband mode (see method RsCmwGprfGen.Source.bbMode) .

return
all_modulations: OFF | ON Comma-separated list of n values, one per frequency/level step, where n 2001. The query returns 2000 results. Each value switches the modulation of a step OFF or ON.

`set`(*index: int*, *modulation: bool*) → None

```
# SCPI: SOURce:GPRF:GENerator<Instance>:LIST:MODulation
driver.source.listPy.modulation.set(index = 1, modulation = False)
```

Defines or queries the 'Mod. On / Off' setting of a selected frequency/level step. The setting is valid only in arbitrary baseband mode (see method RsCmwGprfGen.Source.bbMode) .

param index
integer Number of the frequency/level step in the table Range: 0 to 1999

param modulation
OFF | ON Switch modulation OFF or ON

`set_all`(*all_modulations: List[bool]*) → None

```
# SCPI: SOURce:GPRF:GENerator<Instance>:LIST:MODulation:ALL
driver.source.listPy.modulation.set_all(all_modulations = [True, False, True])
```

Defines or queries the 'Mod. On / Off' setting of all frequency/level steps. The setting is valid only in arbitrary baseband mode (see method RsCmwGprfGen.Source.bbMode) .

param all_modulations
OFF | ON Comma-separated list of n values, one per frequency/level step, where n 2001. The query returns 2000 results. Each value switches the modulation of a step OFF or ON.

6.3.4.8 Reenabling

SCPI Commands :

```
SOURce:GPRF:GENerator<Instance>:LIST:REENabling
SOURce:GPRF:GENerator<Instance>:LIST:REENabling:ALL
```

`class ReenablingCls`

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

`get`(*index: int*) → bool

```
# SCPI: SOURce:GPRF:GENerator<Instance>:LIST:REENabling
value: bool = driver.source.listPy.reenabling.get(index = 1)
```

Defines or queries the 'Reenable On / Off' setting of a selected frequency/level step. The setting is valid if the list increment is enabled by a measurement (see method RsCmwGprfGen.Source.ListPy.Increment.Enabling.value) .

param index
integer Number of the frequency/level step in the table Range: 0 to 1999

return
reenabling: OFF | ON Disable/enable retriggered frequency/level steps

get_all() → List[bool]

```
# SCPI: SOURce:GPRF:GENerator<Instance>:LIST:REENabling:ALL
value: List[bool] = driver.source.listPy.reenabling.get_all()
```

Defines or queries the 'Reenable On / Off' setting of all frequency/level steps. The setting is valid if the list increment is enabled by a measurement (see method RsCmwGprfGen.Source.ListPy.Increment.Enabling.value) .

return
all_reenables: OFF | ON Comma-separated list of n values, one per frequency/level step, where n 2001. The query returns 2000 results. Each value disables/enables a retriggered frequency/level step.

set(*index: int*, *reenabling: bool*) → None

```
# SCPI: SOURce:GPRF:GENerator<Instance>:LIST:REENabling
driver.source.listPy.reenabling.set(index = 1, reenabling = False)
```

Defines or queries the 'Reenable On / Off' setting of a selected frequency/level step. The setting is valid if the list increment is enabled by a measurement (see method RsCmwGprfGen.Source.ListPy.Increment.Enabling.value) .

param index
integer Number of the frequency/level step in the table Range: 0 to 1999

param reenabling
OFF | ON Disable/enable retriggered frequency/level steps

set_all(*all_reenables: List[bool]*) → None

```
# SCPI: SOURce:GPRF:GENerator<Instance>:LIST:REENabling:ALL
driver.source.listPy.reenabling.set_all(all_reenables = [True, False, True])
```

Defines or queries the 'Reenable On / Off' setting of all frequency/level steps. The setting is valid if the list increment is enabled by a measurement (see method RsCmwGprfGen.Source.ListPy.Increment.Enabling.value) .

param all_reenables
OFF | ON Comma-separated list of n values, one per frequency/level step, where n 2001. The query returns 2000 results. Each value disables/enables a retriggered frequency/level step.

6.3.4.9 RfLevel

SCPI Commands :

```
SOURce:GPRF:GENerator<Instance>:LIST:RFLevel
SOURce:GPRF:GENerator<Instance>:LIST:RFLevel:ALL
```

class RfLevelCls

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

get(*index: int*) → float

```
# SCPI: SOURce:GPRF:GENerator<Instance>:LIST:RFLevel
value: float or bool = driver.source.listPy.rfLevel.get(index = 1)
```

Defines or queries the level of the frequency/level step number <Index>.

param index
integer Number of the frequency/level step in the table. Range: 0 to 1999

return
level: (float or boolean) numeric | ON | OFF Level of the frequency/level step Range: Depends on the instrument model, the connector and other settings; please notice the ranges quoted in the data sheet , Unit: dBm ON | OFF enables or disables the frequency/level step.

get_all() → List[float]

```
# SCPI: SOURce:GPRF:GENerator<Instance>:LIST:RFLevel:ALL
value: List[float or bool] = driver.source.listPy.rfLevel.get_all()
```

Defines the levels of all frequency/level steps.

return
all_levels: (float or boolean items) numeric | ON | OFF Comma-separated list of n values, one per frequency/level step, where n 2001. The query returns 2000 results. Range: Depends on the instrument model, the connector and other settings; please notice the ranges quoted in the data sheet , Unit: dBm ON | OFF enables or disables the frequency/level step.

set(*index: int*, *level: float*) → None

```
# SCPI: SOURce:GPRF:GENerator<Instance>:LIST:RFLevel
driver.source.listPy.rfLevel.set(index = 1, level = 1.0)
```

Defines or queries the level of the frequency/level step number <Index>.

param index
integer Number of the frequency/level step in the table. Range: 0 to 1999

param level
(float or boolean) numeric | ON | OFF Level of the frequency/level step Range: Depends on the instrument model, the connector and other settings; please notice the ranges quoted in the data sheet , Unit: dBm ON | OFF enables or disables the frequency/level step.

set_all(*all_levels: List[float]*) → None

```
# SCPI: SOURce:GPRF:GENerator<Instance>:LIST:RFLevel:ALL
driver.source.listPy.rfLevel.set_all(all_levels = [1.1, True, 2.2, False, 3.3])
```

Defines the levels of all frequency/level steps.

param all_levels
(float or boolean items) numeric | ON | OFF Comma-separated list of n values, one per frequency/level step, where n 2001. The query returns 2000 results. Range: Depends on the instrument model, the connector and other settings; please notice the ranges quoted in the data sheet , Unit: dBm ON | OFF enables or disables the frequency/level step.

6.3.4.10 Rlist

SCPI Command :

```
SOURce:GPRF:GENerator<Instance>:LIST:RLISt
```

class RlistCls

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

set() → None

```
# SCPI: SOURce:GPRF:GENerator<Instance>:LIST:RLISt
driver.source.listPy.rlist.set()
```

Restarts the list generator at the first frequency/level step. This command provides a fast alternative to a complete restart of the list generator (turn generator off and on again) . The active list index can be queried using method RsCmwGprfGen.Source.ListPy.aindex.

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

```
# SCPI: SOURce:GPRF:GENerator<Instance>:LIST:RLISt
driver.source.listPy.rlist.set_with_opc()
```

Restarts the list generator at the first frequency/level step. This command provides a fast alternative to a complete restart of the list generator (turn generator off and on again) . The active list index can be queried using method RsCmwGprfGen.Source.ListPy.aindex.

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

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

6.3.4.11 SingleCmw

SCPI Command :

```
SOURce:GPRF:GENerator<Instance>:LIST:CMWS:CSET
```

class SingleCmwCls

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

get_cset() → ParameterSetMode

```
# SCPI: SOURce:GPRF:GENerator<Instance>:LIST:CMWS:CSET
value: enums.ParameterSetMode = driver.source.listPy.singleCmw.get_cset()
```

No command help available

return
cmws_connector_set: No help available

set_cset(*cmws_connector_set: ParameterSetMode*) → None

```
# SCPI: SOURce:GPRF:GENerator<Instance>:LIST:CMWS:CSET
driver.source.listPy.singleCmw.set_cset(cmws_connector_set = enums.
↪ParameterSetMode.GLOBal)
```

No command help available

param cmws_connector_set
No help available

Cloning the Group

```
# Create a clone of the original group, that exists independently
group2 = driver.source.listPy.singleCmw.clone()
```

Subgroups

6.3.4.11.1 Usage

class UsageCls

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

Cloning the Group

```
# Create a clone of the original group, that exists independently
group2 = driver.source.listPy.singleCmw.usage.clone()
```

Subgroups

6.3.4.11.1.1 Tx

SCPI Command :

```
SOURce:GPRF:GENerator<Instance>:LIST:CMWS:USAGe:TX
```

class TxCls

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

get(*index: int*) → List[bool]

```
# SCPI: SOURce:GPRF:GENerator<Instance>:LIST:CMWS:USAGe:TX
value: List[bool] = driver.source.listPy.singleCmw.usage.tx.get(index = 1)
```

No command help available

param index
No help available

return
usage: No help available

set(*index: int*, *usage: List[bool]*) → None

```
# SCPI: SOURce:GPRF:GENerator<Instance>:LIST:CMWS:USAGe:TX
driver.source.listPy.singleCmw.usage.tx.set(index = 1, usage = [True, False,␣
↪True])
```

No command help available

param index
No help available

param usage
No help available

6.3.4.12 Slist

SCPI Command :

```
SOURce:GPRF:GENerator<Instance>:LIST:SLISt
```

class SlistCls

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

set() → None

```
# SCPI: SOURce:GPRF:GENerator<Instance>:LIST:SLISt
driver.source.listPy.slist.set()
```

This command initiates the list cycling in CONTinuous repetition mode (see method RsCmwGprfGen.Source.ListPy.repetition) , for lists incremented by dwell time or ARB file marker (see method RsCmwGprfGen.Source.ListPy.Increment.value) and with 'Manual' list increment enabling (see method

RsCmwGprfGen.Source.ListPy.Increment.Enabling.value) . The active list index can be queried using method RsCmwGprfGen.Source.ListPy.aindex.

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

```
# SCPI: SOURce:GPRF:GENerator<Instance>:LIST:SLISt
driver.source.listPy.slist.set_with_opc()
```

This command initiates the list cycling in CONTinuous repetition mode (see method RsCmwGprfGen.Source.ListPy.repetition) , for lists incremented by dwell time or ARB file marker (see method RsCmwGprfGen.Source.ListPy.Increment.value) and with 'Manual' list increment enabling (see method RsCmwGprfGen.Source.ListPy.Increment.Enabling.value) . The active list index can be queried using method RsCmwGprfGen.Source.ListPy.aindex.

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

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

6.3.4.13 Sstop

SCPI Command :

```
SOURce:GPRF:GENerator<Instance>:LIST:SSTop
```

class SstopCls

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

class GetStruct

Response structure. Fields:

- Start_Index: int: integer Range: 0 to min{StopIndex,1999}
- Stop_Index: int: integer Range: StartIndex to 1999

get() → GetStruct

```
# SCPI: SOURce:GPRF:GENerator<Instance>:LIST:SSTop
value: GetStruct = driver.source.listPy.sstop.get()
```

Defines the first and last generated frequency/level steps in list mode.

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

set(*start_index: int*, *stop_index: int*, *goto_index: int = None*) → None

```
# SCPI: SOURce:GPRF:GENerator<Instance>:LIST:SSTop
driver.source.listPy.sstop.set(start_index = 1, stop_index = 1, goto_index = 1)
```

Defines the first and last generated frequency/level steps in list mode.

param start_index
integer Range: 0 to min{StopIndex,1999}

param stop_index
integer Range: StartIndex to 1999

param goto_index
integer The start index for all but the first generator cycle in continuous mode. See also method RsCmwGprfGen.Source.ListPy.goto. Range: StartIndex to StopIndex

6.3.5 RfSettings

SCPI Commands :

```
SOURce:GPRF:GENerator<Instance>:RFSettings:DGAin
SOURce:GPRF:GENerator<Instance>:RFSettings:PEPower
SOURce:GPRF:GENerator<Instance>:RFSettings:EATTenuation
SOURce:GPRF:GENerator<Instance>:RFSettings:FREQuency
SOURce:GPRF:GENerator<Instance>:RFSettings:LEVel
```

class RfSettingsCls

RfSettings commands group definition. 5 total commands, 0 Subgroups, 5 group commands

get_dgain() → float

```
# SCPI: SOURce:GPRF:GENerator<Instance>:RFSettings:DGAin
value: float = driver.source.rfSettings.get_dgain()
```

Defines the digital gain of the RF generator.

return
digital_gain: numeric Unit: dB

get_eattenuation() → float

```
# SCPI: SOURce:GPRF:GENerator<Instance>:RFSettings:EATTenuation
value: float = driver.source.rfSettings.get_eattenuation()
```

Defines an external attenuation (or gain, if the value is negative) , to be applied to the output connector.

return
ext_rf_out_att: numeric Range: -50 dB to 90 dB, Unit: dB

get_frequency() → float

```
# SCPI: SOURce:GPRF:GENerator<Instance>:RFSettings:FREQuency
value: float = driver.source.rfSettings.get_frequency()
```

Sets the frequency of the unmodulated RF carrier. For the supported frequency range, see 'Frequency ranges'.

return
frequency: numeric Unit: Hz

get_level() → float

```
# SCPI: SOURce:GPRF:GENerator<Instance>:RFSettings:LEVel
value: float = driver.source.rfSettings.get_level()
```

Sets the base RMS level of the RF generator.

return
level: numeric Range: Please notice the ranges quoted in the data sheet. , Unit: dBm

get_pe_power() → float

```
# SCPI: SOURce:GPRF:GENerator<Instance>:RFSettings:PEPower
value: float = driver.source.rfSettings.get_pe_power()
```

Queries the peak envelope power.

return
peak_envelope_pow: float Unit: dBm

set_dgain(*digital_gain: float*) → None

```
# SCPI: SOURce:GPRF:GENerator<Instance>:RFSettings:DGAin
driver.source.rfSettings.set_dgain(digital_gain = 1.0)
```

Defines the digital gain of the RF generator.

param digital_gain
numeric Unit: dB

set_eattenuation(*ext_rf_out_att: float*) → None

```
# SCPI: SOURce:GPRF:GENerator<Instance>:RFSettings:EATTenuation
driver.source.rfSettings.set_eattenuation(ext_rf_out_att = 1.0)
```

Defines an external attenuation (or gain, if the value is negative) , to be applied to the output connector.

param ext_rf_out_att
numeric Range: -50 dB to 90 dB, Unit: dB

set_frequency(*frequency: float*) → None

```
# SCPI: SOURce:GPRF:GENerator<Instance>:RFSettings:FREQuency
driver.source.rfSettings.set_frequency(frequency = 1.0)
```

Sets the frequency of the unmodulated RF carrier. For the supported frequency range, see 'Frequency ranges'.

param frequency
numeric Unit: Hz

set_level(*level: float*) → None

```
# SCPI: SOURce:GPRF:GENerator<Instance>:RFSettings:LEVel
driver.source.rfSettings.set_level(level = 1.0)
```

Sets the base RMS level of the RF generator.

param level
numeric Range: Please notice the ranges quoted in the data sheet. , Unit: dBm

6.3.6 Sequencer

SCPI Commands :

```
SOURce:GPRF:GENerator<Instance>:SEQuencer:REPetition
SOURce:GPRF:GENerator<Instance>:SEQuencer:NREPetition
SOURce:GPRF:GENerator<Instance>:SEQuencer:RCOunt
SOURce:GPRF:GENerator<Instance>:SEQuencer:SIGNal
SOURce:GPRF:GENerator<Instance>:SEQuencer:CENTry
SOURce:GPRF:GENerator<Instance>:SEQuencer:UOPTions
```

class SequencerCls

Sequencer commands group definition. 92 total commands, 9 Subgroups, 6 group commands

get_centry() → int

```
# SCPI: SOURce:GPRF:GENerator<Instance>:SEQuencer:CENTry
value: int = driver.source.sequencer.get_centry()
```

Queries the index of the processed entry. The remote query takes between 2 ms and 3 ms, which introduces an uncertainty to the results.

return
current_entry: decimal If the sequencer is not running, NAV is returned.

get_nrepetition() → int

```
# SCPI: SOURce:GPRF:GENerator<Instance>:SEQuencer:NREPetition
value: int = driver.source.sequencer.get_nrepetition()
```

No command help available

return
num_of_rep: No help available

get_rcount() → int

```
# SCPI: SOURce:GPRF:GENerator<Instance>:SEQuencer:RCOunt
value: int = driver.source.sequencer.get_rcount()
```

No command help available

return
repcount: No help available

get_repetition() → RepeatMode

```
# SCPI: SOURce:GPRF:GENerator<Instance>:SEQuencer:REPetition
value: enums.RepeatMode = driver.source.sequencer.get_repetition()
```

Defines the repetition mode for the sequencer list.

return
repetition: CONTinuous | SINGle CONTinuous: unlimited repetitions, with cyclic processing SINGle: single execution

get_signal() → bool

```
# SCPI: SOURce:GPRF:GENerator<Instance>:SEQuencer:SIGNal
value: bool = driver.source.sequencer.get_signal()
```

Queries whether a signal is generated or not.

return
signal: OFF | ON

get_uoptions() → str

```
# SCPI: SOURce:GPRF:GENerator<Instance>:SEQuencer:UOPTions
value: str = driver.source.sequencer.get_uoptions()
```

Queries a list of the used software options.

return
used_options: string The string contains a comma-separated list of options. If the sequencer is OFF, NAV is returned. If the sequencer is not OFF but no options are used by the sequencer list, 'None' is returned.

set_nrepetition(*num_of_rep: int*) → None

```
# SCPI: SOURce:GPRF:GENerator<Instance>:SEQuencer:NREPetition
driver.source.sequencer.set_nrepetition(num_of_rep = 1)
```

No command help available

param num_of_rep
No help available

set_repetition(*repetition: RepeatMode*) → None

```
# SCPI: SOURce:GPRF:GENerator<Instance>:SEQuencer:REPetition
driver.source.sequencer.set_repetition(repetition = enums.RepeatMode.CONTinuous)
```

Defines the repetition mode for the sequencer list.

param repetition
CONTinuous | SINGle CONTinuous: unlimited repetitions, with cyclic processing SINGle: single execution

Cloning the Group

```
# Create a clone of the original group, that exists independently
group2 = driver.source.sequencer.clone()
```

Subgroups

6.3.6.1 Apool

SCPI Commands :

```
SOURce:GPRF:GENerator<Instance>:SEQuencer:APOol:VALid
SOURce:GPRF:GENerator<Instance>:SEQuencer:APOol:LOADed
SOURce:GPRF:GENerator<Instance>:SEQuencer:APOol:RREQuired
SOURce:GPRF:GENerator<Instance>:SEQuencer:APOol:RTOTal
SOURce:GPRF:GENerator<Instance>:SEQuencer:APOol:FILE
SOURce:GPRF:GENerator<Instance>:SEQuencer:APOol:REMove
SOURce:GPRF:GENerator<Instance>:SEQuencer:APOol:CLEar
SOURce:GPRF:GENerator<Instance>:SEQuencer:APOol:MINDex
```

class ApoolCls

Apool commands group definition. 30 total commands, 12 Subgroups, 8 group commands

clear() → None

```
# SCPI: SOURce:GPRF:GENerator<Instance>:SEQuencer:APOol:CLEar
driver.source.sequencer.apool.clear()
```

Removes all files from the ARB file pool.

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

```
# SCPI: SOURce:GPRF:GENerator<Instance>:SEQuencer:APOol:CLEar
driver.source.sequencer.apool.clear_with_opc()
```

Removes all files from the ARB file pool.

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

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

get_loaded() → YesNoStatus

```
# SCPI: SOURce:GPRF:GENerator<Instance>:SEQuencer:APOol:LOADed
value: enums.YesNoStatus = driver.source.sequencer.apool.get_loaded()
```

Queries whether the ARB file pool is downloaded to the ARB RAM.

return
loaded: NO | YES

get_mindex() → int

```
# SCPI: SOURce:GPRF:GENerator<Instance>:SEQuencer:APOol:MINDex
value: int = driver.source.sequencer.apool.get_mindex()
```

Queries the highest index of the ARB file pool. The pool contains files with the indices 0 to <MaximumIndex>.

return
maximum_index: decimal Highest index. If the file pool is empty, NAV is returned.

get_rrequired() → float

```
# SCPI: SOURce:GPRF:GENerator<Instance>:SEQuencer:APOol:RREQuired
value: float = driver.source.sequencer.apool.get_rrequired()
```

Queries the amount of RAM required by the ARB files in the pool.

return
ram_required: float Unit: Mbyte

get_rtotal() → float

```
# SCPI: SOURce:GPRF:GENerator<Instance>:SEQuencer:APOol:RTOTal
value: float = driver.source.sequencer.apool.get_rtotal()
```

Queries the amount of RAM available for ARB files.

return
ram_total: float Unit: Mbyte

get_valid() → YesNoStatus

```
# SCPI: SOURce:GPRF:GENerator<Instance>:SEQuencer:APOol:VALid
value: enums.YesNoStatus = driver.source.sequencer.apool.get_valid()
```

Queries whether the ARB file pool is valid.

return
valid: NO | YES

set_file(*arb_file: str*) → None

```
# SCPI: SOURce:GPRF:GENerator<Instance>:SEQuencer:APOol:FILE
driver.source.sequencer.apool.set_file(arb_file = 'abc')
```

Adds an ARB file to the ARB file pool.

param arb_file
string Path and filename Example: ‘@WAVEFORM/myARBfile.wv’

set_remove(*indices: List[int]*) → None

```
# SCPI: SOURce:GPRF:GENerator<Instance>:SEQuencer:APOol:REMove
driver.source.sequencer.apool.set_remove(indices = [1, 2, 3])
```

Removes selected files from the ARB file pool.

param indices
integer Indices of the files to be removed. You can specify a single index or a comma-separated list of indices.

Cloning the Group

```
# Create a clone of the original group, that exists independently
group2 = driver.source.sequencer.apool.clone()
```

Subgroups

6.3.6.1.1 CrcProtect

SCPI Commands :

```
SOURce:GPRF:GENerator<Instance>:SEQuencer:APOol:CRCProtect:ALL
SOURce:GPRF:GENerator<Instance>:SEQuencer:APOol:CRCProtect
```

class CrcProtectCls

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

get(*index: int*) → YesNoStatus

```
# SCPI: SOURce:GPRF:GENerator<Instance>:SEQuencer:APOol:CRCProtect
value: enums.YesNoStatus = driver.source.sequencer.apool.crcProtect.get(index =␣
↪1)
```

Queries whether the ARB file with the specified <Index> contains a CRC checksum.

param index
integer

return
crc_protection: NO | YES

get_all() → List[YesNoStatus]

```
# SCPI: SOURce:GPRF:GENerator<Instance>:SEQuencer:APOol:CRCProtect:ALL
value: List[enums.YesNoStatus] = driver.source.sequencer.apool.crcProtect.get_
↪all()
```

Queries whether the ARB files in the file pool contain CRC checksums.

return
crc_protection: NO | YES Comma-separated list of values, one value per file

6.3.6.1.2 Download

SCPI Command :

```
SOURce:GPRF:GENerator<Instance>:SEQuencer:APOol:DOWNload
```

class DownloadCls

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

set() → None

```
# SCPI: SOURce:GPRF:GENerator<Instance>:SEQuencer:APOol:DOWNload
driver.source.sequencer.apool.download.set()
```

Downloads the ARB files from the file pool to the ARB RAM.

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

```
# SCPI: SOURce:GPRF:GENerator<Instance>:SEQuencer:APOol:DOWNload
driver.source.sequencer.apool.download.set_with_opc()
```

Downloads the ARB files from the file pool to the ARB RAM.

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

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

6.3.6.1.3 Duration

SCPI Commands :

```
SOURce:GPRF:GENerator<Instance>:SEQuencer:APOol:DURation:ALL
SOURce:GPRF:GENerator<Instance>:SEQuencer:APOol:DURation
```

class DurationCls

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

get(*index: int*) → float

```
# SCPI: SOURce:GPRF:GENerator<Instance>:SEQuencer:APOol:DURation
value: float = driver.source.sequencer.apool.duration.get(index = 1)
```

Queries the duration of the ARB file with the specified <Index>.

param index
integer

return
duration: float Unit: s

get_all() → List[float]

```
# SCPI: SOURce:GPRF:GENerator<Instance>:SEQuencer:APOol:DURation:ALL
value: List[float] = driver.source.sequencer.apool.duration.get_all()
```

Queries the durations of the ARB files in the file pool.

return
duration: float Comma-separated list of values, one value per file Unit: s

6.3.6.1.4 Paratio

SCPI Commands :

```
SOURce:GPRF:GENerator<Instance>:SEQuencer:APOol:PARatio:ALL
SOURce:GPRF:GENerator<Instance>:SEQuencer:APOol:PARatio
```

class ParatioCls

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

get(*index: int*) → float

```
# SCPI: SOURce:GPRF:GENerator<Instance>:SEQuencer:APOol:PARatio
value: float = driver.source.sequencer.apool.paratio.get(index = 1)
```

Queries the peak to average ratio of the ARB file with the specified <Index>.

param index
integer

return
peak_avg_ratio: float Unit: dB

get_all() → List[float]

```
# SCPI: SOURce:GPRF:GENerator<Instance>:SEQuencer:APOol:PARatio:ALL
value: List[float] = driver.source.sequencer.apool.paratio.get_all()
```

Queries the peak to average ratios of the ARB files in the file pool.

return
peak_avg_ratio: float Comma-separated list of values, one value per file Unit: dB

6.3.6.1.5 Path

SCPI Commands :

```
SOURce:GPRF:GENerator<Instance>:SEQuencer:APOol:PATH:ALL
SOURce:GPRF:GENerator<Instance>:SEQuencer:APOol:PATH
```

class PathCls

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

get(*index: int*) → str

```
# SCPI: SOURce:GPRF:GENerator<Instance>:SEQuencer:APOol:PATH
value: str = driver.source.sequencer.apool.path.get(index = 1)
```

Queries the path and file name of the ARB file with the specified <Index>.

param index
integer

return
full_path_name: string Complete path and name of the file.

get_all() → List[str]

```
# SCPI: SOURce:GPRF:GENerator<Instance>:SEQuencer:APOol:PATH:ALL
value: List[str] = driver.source.sequencer.apool.path.get_all()
```

Queries the path and file name of the ARB files in the file pool.

return
path: string Comma-separated list of strings, one string per file Each string contains the complete path and name of a file.

6.3.6.1.6 Poffset

SCPI Commands :

```
SOURce:GPRF:GENerator<Instance>:SEQuencer:APOol:POFFset:ALL
SOURce:GPRF:GENerator<Instance>:SEQuencer:APOol:POFFset
```

class PoffsetCls

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

get(*index: int*) → float

```
# SCPI: SOURce:GPRF:GENerator<Instance>:SEQuencer:APOol:POFFset
value: float = driver.source.sequencer.apool.poffset.get(index = 1)
```

Queries the peak offset of the ARB file with the specified <Index>.

param index
integer

return
peak_offset: float Unit: dB

get_all() → List[float]

```
# SCPI: SOURce:GPRF:GENerator<Instance>:SEQuencer:APOol:POFFset:ALL
value: List[float] = driver.source.sequencer.apool.poffset.get_all()
```

Queries the peak offsets of the ARB files in the file pool.

return
peak_offset: float Comma-separated list of values, one value per file Unit: dB

6.3.6.1.7 Reliability

SCPI Command :

```
SOURce:GPRF:GENerator<Instance>:SEQuencer:APOol:RELiability
```

class ReliabilityCls

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

get(*index: int*) → int

```
# SCPI: SOURce:GPRF:GENerator<Instance>:SEQuencer:APOol:RELiability
value: int = driver.source.sequencer.apool.reliability.get(index = 1)
```

Queries the reliability indicator for the ARB file with the specified <Index>. For possible values, see 'Reliability indicator'.

param index
integer

return
reliability: decimal Reliability indicator

6.3.6.1.8 Rmessage

SCPI Commands :

```
SOURce:GPRF:GENerator<Instance>:SEQuencer:APOol:RMESsage:ALL
SOURce:GPRF:GENerator<Instance>:SEQuencer:APOol:RMESsage
```

class RmessageCls

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

get(*index: int*) → str

```
# SCPI: SOURce:GPRF:GENerator<Instance>:SEQuencer:APOol:RMESsage
value: str = driver.source.sequencer.apool.rmessage.get(index = 1)
```

Queries the reliability message for the ARB file with the specified <Index>. For possible values, see 'Reliability indicator'.

param index
integer

return
reliability_msg: string Reliability message

get_all() → List[str]

```
# SCPI: SOURce:GPRF:GENerator<Instance>:SEQuencer:APOol:RMESsage:ALL
value: List[str] = driver.source.sequencer.apool.rmessage.get_all()
```

Queries the reliability messages for all ARB files in the file pool. For possible values, see 'Reliability indicator'.

return
reliability_msg: string Comma-separated list of strings One string per file, from index 0 to index n

6.3.6.1.9 Roption

SCPI Commands :

```
SOURce:GPRF:GENerator<Instance>:SEQuencer:APOol:ROPTion:ALL
SOURce:GPRF:GENerator<Instance>:SEQuencer:APOol:ROPTion
```

class RoptionCls

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

get(*index: int*) → str

```
# SCPI: SOURce:GPRF:GENerator<Instance>:SEQuencer:APOol:ROPTion
value: str = driver.source.sequencer.apool.roption.get(index = 1)
```

Queries the options required by the ARB file with the specified <Index>.

param index
integer

return
required_options: string

get_all() → List[str]

```
# SCPI: SOURce:GPRF:GENerator<Instance>:SEQuencer:APOol:ROPTion:ALL
value: List[str] = driver.source.sequencer.apool.roption.get_all()
```

Queries the options required by the ARB files in the file pool.

return
required_options: string Comma-separated list of strings, one string per file

6.3.6.1.10 Samples

SCPI Commands :

```
SOURce:GPRF:GENerator<Instance>:SEQuencer:APOol:SAMPles:ALL
SOURce:GPRF:GENerator<Instance>:SEQuencer:APOol:SAMPles
```

class SamplesCls

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

get(*index: int*) → int

```
# SCPI: SOURce:GPRF:GENerator<Instance>:SEQuencer:APOol:SAMPles
value: int = driver.source.sequencer.apool.samples.get(index = 1)
```

Queries the number of samples in the ARB file with the specified <Index>.

param index
integer

return
samples: decimal Number of samples

get_all() → List[int]

```
# SCPI: SOURce:GPRF:GENerator<Instance>:SEQuencer:APOol:SAMPles:ALL
value: List[int] = driver.source.sequencer.apool.samples.get_all()
```

Queries the numbers of samples in the ARB files of the file pool.

return
samples: decimal Comma-separated list of values, one value per file

6.3.6.1.11 SymbolRate

SCPI Commands :

```
SOURce:GPRF:GENerator<Instance>:SEQuencer:APOol:SRATe:ALL
SOURce:GPRF:GENerator<Instance>:SEQuencer:APOol:SRATe
```

class SymbolRateCls

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

get(*index: int*) → float

```
# SCPI: SOURce:GPRF:GENerator<Instance>:SEQuencer:APOol:SRATe
value: float = driver.source.sequencer.apool.symbolRate.get(index = 1)
```

Queries the sample rate of the ARB file with the specified <Index>.

param index
integer

return
sample_rate: float Unit: Samples per second

get_all() → List[float]

```
# SCPI: SOURce:GPRF:GENerator<Instance>:SEQuencer:APOol:SRATe:ALL
value: List[float] = driver.source.sequencer.apool.symbolRate.get_all()
```

Queries the sample rates of the ARB files in the file pool.

return
sample_rate: float Comma-separated list of values, one value per file Unit: Samples per second

6.3.6.1.12 Waveform

SCPI Commands :

```
SOURce:GPRF:GENerator<Instance>:SEQuencer:APOol:WAVeform:ALL
SOURce:GPRF:GENerator<Instance>:SEQuencer:APOol:WAVeform
```

class WaveformCls

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

get(*index: int*) → str

```
# SCPI: SOURce:GPRF:GENerator<Instance>:SEQuencer:APOol:WAVeform
value: str = driver.source.sequencer.apool.waveform.get(index = 1)
```

Queries the name of the ARB file with the specified <Index>.

param index
integer

return
waveform: string File name (without path) .

get_all() → List[str]

```
# SCPI: SOURce:GPRF:GENerator<Instance>:SEQuencer:APOol:WAVeform:ALL
value: List[str] = driver.source.sequencer.apool.waveform.get_all()
```

Queries the names of the ARB files in the file pool.

return
waveform: string Comma-separated list of strings One string per file, from index 0 to index n

6.3.6.2 Dtone

SCPI Command :

```
SOURce:GPRF:GENerator<Instance>:SEQuencer:DTONe:RATio
```

class DtoneCls

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

get_ratio() → float

```
# SCPI: SOURce:GPRF:GENerator<Instance>:SEQuencer:DTONe:RATio
value: float = driver.source.sequencer.dtone.get_ratio()
```

No command help available

return
ratio: No help available

set_ratio(*ratio: float*) → None

```
# SCPI: SOURce:GPRF:GENerator<Instance>:SEQuencer:DTONe:RATio
driver.source.sequencer.dtone.set_ratio(ratio = 1.0)
```

No command help available

param ratio
No help available

Cloning the Group

```
# Create a clone of the original group, that exists independently
group2 = driver.source.sequencer.dtone.clone()
```

Subgroups

6.3.6.2.1 Ofrequency<FrequencySource>

RepCap Settings

```
# Range: Src1 .. Src2
rc = driver.source.sequencer.dtone.ofrequency.repcap_frequencySource_get()
driver.source.sequencer.dtone.ofrequency.repcap_frequencySource_set(repcap.
↪FrequencySource.Src1)
```

SCPI Command :

```
SOURce:GPRF:GENerator<Instance>:SEQuencer:DTONe:OFRequency<source>
```

class OfrequencyCls

Ofrequency commands group definition. 1 total commands, 0 Subgroups, 1 group commands Repeated Capability: FrequencySource, default value after init: FrequencySource.Src1

get(*frequencySource=FrequencySource.Default*) → float

```
# SCPI: SOURce:GPRF:GENerator<Instance>:SEQuencer:DTONe:OFRequency<source>
value: float = driver.source.sequencer.dtone.ofrequency.get(frequencySource =␣
↪repcap.FrequencySource.Default)
```

No command help available

param frequencySource
optional repeated capability selector. Default value: Src1 (settable in the interface 'Ofrequency')

return
frequency: No help available

set(*frequency: float*, *frequencySource=FrequencySource.Default*) → None

```
# SCPI: SOURce:GPRF:GENerator<Instance>:SEQuencer:DTONe:OFRequency<source>
driver.source.sequencer.dtone.ofrequency.set(frequency = 1.0, frequencySource =␣
↪repcap.FrequencySource.Default)
```

No command help available

param frequency
No help available

param frequencySource
optional repeated capability selector. Default value: Src1 (settable in the interface 'Ofrequency')

Cloning the Group

```
# Create a clone of the original group, that exists independently
group2 = driver.source.sequencer.dtone.ofrequency.clone()
```

6.3.6.3 ListPy

SCPI Commands :

```
SOURce:GPRF:GENerator<Instance>:SEQuencer:LIST:CREate
SOURce:GPRF:GENerator<Instance>:SEQuencer:LIST:INDex
SOURce:GPRF:GENerator<Instance>:SEQuencer:LIST:MINDex
```

class ListPyCls

ListPy commands group definition. 44 total commands, 14 Subgroups, 3 group commands

get_index() → int

```
# SCPI: SOURce:GPRF:GENerator<Instance>:SEQuencer:LIST:INDex
value: int = driver.source.sequencer.listPy.get_index()
```

Selects an entry of the sequencer list. Some other commands use this setting.

return
current_index: integer Index of the selected list entry

get_mindex() → int

```
# SCPI: SOURce:GPRF:GENerator<Instance>:SEQuencer:LIST:MINDex
value: int = driver.source.sequencer.listPy.get_mindex()
```

Queries the highest index of the sequencer list. The list contains entries with the indices 0 to <MaximumIndex>.

return
maximum_index: decimal

set_create(*entries: float*) → None

```
# SCPI: SOURce:GPRF:GENerator<Instance>:SEQuencer:LIST:CREate
driver.source.sequencer.listPy.set_create(entries = 1.0)
```

Deletes all entries of the sequencer list and creates the defined number of new entries with default settings.

param entries
numeric Number of entries to be created

set_index(*current_index: int*) → None

```
# SCPI: SOURce:GPRF:GENerator<Instance>:SEQuencer:LIST:INDex
driver.source.sequencer.listPy.set_index(current_index = 1)
```

Selects an entry of the sequencer list. Some other commands use this setting.

param current_index
integer Index of the selected list entry

Cloning the Group

```
# Create a clone of the original group, that exists independently
group2 = driver.source.sequencer.listPy.clone()
```

Subgroups

6.3.6.3.1 Acycles

SCPI Commands :

```
SOURce:GPRF:GENerator<Instance>:SEQuencer:LIST:ACYCles
SOURce:GPRF:GENerator<Instance>:SEQuencer:LIST:ACYCles:ALL
```

class AcyclesCls

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

get(*index: int*) → int

```
# SCPI: SOURce:GPRF:GENerator<Instance>:SEQuencer:LIST:ACYCles
value: int = driver.source.sequencer.listPy.acycles.get(index = 1)
```

Defines or queries the number of ARB cycles for the sequencer list entry with the selected <Index>.

param index
integer

return
arb_cycles: numeric Range: 1 to 10000

get_all() → List[int]

```
# SCPI: SOURce:GPRF:GENerator<Instance>:SEQuencer:LIST:ACYCles:ALL
value: List[int] = driver.source.sequencer.listPy.acycles.get_all()
```

Defines the ARB cycles for all sequencer list entries.

return
arb_cycles: numeric Comma-separated list of values, one value per list entry Range: 1 to 10000

set(*index: int*, *arb_cycles: int*) → None

```
# SCPI: SOURce:GPRF:GENerator<Instance>:SEQuencer:LIST:ACYCles
driver.source.sequencer.listPy.acycles.set(index = 1, arb_cycles = 1)
```

Defines or queries the number of ARB cycles for the sequencer list entry with the selected <Index>.

param index
integer

param arb_cycles
numeric Range: 1 to 10000

set_all(*arb_cycles: List[int]*) → None

```
# SCPI: SOURce:GPRF:GENerator<Instance>:SEQuencer:LIST:ACYCles:ALL
driver.source.sequencer.listPy.acycles.set_all(arb_cycles = [1, 2, 3])
```

Defines the ARB cycles for all sequencer list entries.

param arb_cycles
numeric Comma-separated list of values, one value per list entry Range: 1 to 10000

6.3.6.3.2 Dgain

SCPI Commands :

```
SOURce:GPRF:GENerator<Instance>:SEQuencer:LIST:DGAin
SOURce:GPRF:GENerator<Instance>:SEQuencer:LIST:DGAin:ALL
```

class DgainCls

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

get(*index: int*) → float

```
# SCPI: SOURce:GPRF:GENerator<Instance>:SEQuencer:LIST:DGAin
value: float = driver.source.sequencer.listPy.dgain.get(index = 1)
```

Defines or queries the digital gain for the sequencer list entry with the selected <Index>.

param index
integer

return
digital_gain: numeric Unit: dB

get_all() → List[float]

```
# SCPI: SOURce:GPRF:GENerator<Instance>:SEQuencer:LIST:DGAin:ALL
value: List[float] = driver.source.sequencer.listPy.dgain.get_all()
```

Defines the digital gains for all sequencer list entries.

return
digital_gain: numeric Comma-separated list of values, one value per list entry Unit: dB

set(*index: int*, *digital_gain: float*) → None

```
# SCPI: SOURce:GPRF:GENerator<Instance>:SEQuencer:LIST:DGAin
driver.source.sequencer.listPy.dgain.set(index = 1, digital_gain = 1.0)
```

Defines or queries the digital gain for the sequencer list entry with the selected <Index>.

param index
integer

param digital_gain
numeric Unit: dB

set_all(*digital_gain: List[float]*) → None

```
# SCPI: SOURce:GPRF:GENerator<Instance>:SEQuencer:LIST:DGAin:ALL
driver.source.sequencer.listPy.dgain.set_all(digital_gain = [1.1, 2.2, 3.3])
```

Defines the digital gains for all sequencer list entries.

param digital_gain
numeric Comma-separated list of values, one value per list entry Unit: dB

6.3.6.3.3 Dtime

SCPI Commands :

```
SOURce:GPRF:GENerator<Instance>:SEQuencer:LIST:DTIMe
SOURce:GPRF:GENerator<Instance>:SEQuencer:LIST:DTIMe:ALL
```

class DtimeCls

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

get(*index: int*) → float

```
# SCPI: SOURce:GPRF:GENerator<Instance>:SEQuencer:LIST:DTIMe
value: float = driver.source.sequencer.listPy.dtime.get(index = 1)
```

Defines or queries the dwell time for the sequencer list entry with the selected <Index>.

param index
integer

return
dwell_time: numeric Range: 200E-6 s to 20 s, Unit: s

get_all() → List[float]

```
# SCPI: SOURce:GPRF:GENerator<Instance>:SEQuencer:LIST:DTIMe:ALL
value: List[float] = driver.source.sequencer.listPy.dtime.get_all()
```

Defines the dwell times for all sequencer list entries.

return
dwell_time: numeric Comma-separated list of values, one value per list entry Range: 200E-6 s to 20 s, Unit: s

set(*index: int*, *dwell_time: float*) → None

```
# SCPI: SOURce:GPRF:GENerator<Instance>:SEQuencer:LIST:DTIMe
driver.source.sequencer.listPy.dtime.set(index = 1, dwell_time = 1.0)
```

Defines or queries the dwell time for the sequencer list entry with the selected <Index>.

param index
integer

param dwell_time
numeric Range: 200E-6 s to 20 s, Unit: s

set_all(*dwell_time: List[float]*) → None

```
# SCPI: SOURce:GPRF:GENerator<Instance>:SEQuencer:LIST:DTIMe:ALL
driver.source.sequencer.listPy.dtime.set_all(dwell_time = [1.1, 2.2, 3.3])
```

Defines the dwell times for all sequencer list entries.

param dwell_time
numeric Comma-separated list of values, one value per list entry Range: 200E-6 s to 20 s, Unit: s

6.3.6.3.4 Entry

SCPI Command :

```
SOURce:GPRF:GENerator<Instance>:SEQuencer:LIST:ENTRy:DELete
```

class EntryCls

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

delete(*index: int = None*) → None

```
# SCPI: SOURce:GPRF:GENerator<Instance>:SEQuencer:LIST:ENTRy:DELete
driver.source.sequencer.listPy.entry.delete(index = 1)
```

Deletes the selected entry from the sequencer list. You can specify <Index> to select that entry. Or you can select an entry via method RsCmwGprfGen.Source.Sequencer.ListPy.index. After the deletion, the selection moves to the next entry, if possible. Otherwise, it moves to the previous entry.

param index
integer Index of the entry to be deleted

Cloning the Group

```
# Create a clone of the original group, that exists independently
group2 = driver.source.sequencer.listPy.entry.clone()
```

Subgroups

6.3.6.3.4.1 Call

SCPI Command :

```
SOURce:GPRF:GENerator<Instance>:SEQuencer:LIST:ENTRy:CALL
```

class CallCls

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

set() → None

```
# SCPI: SOURce:GPRF:GENerator<Instance>:SEQuencer:LIST:ENTRy:CALL
driver.source.sequencer.listPy.entry.call.set()
```

Deletes all entries of the sequencer list and creates an entry with default settings.

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

```
# SCPI: SOURce:GPRF:GENerator<Instance>:SEQuencer:LIST:ENTRy:CALL
driver.source.sequencer.listPy.entry.call.set_with_opc()
```

Deletes all entries of the sequencer list and creates an entry with default settings.

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

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

6.3.6.3.4.2 Insert

SCPI Command :

```
SOURce:GPRF:GENerator<Instance>:SEQuencer:LIST:ENTRy:INSert
```

class InsertCls

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

set(*after_index: int = None*) → None

```
# SCPI: SOURce:GPRF:GENerator<Instance>:SEQuencer:LIST:ENTRy:INSert
driver.source.sequencer.listPy.entry.insert.set(after_index = 1)
```

Inserts a new entry into the sequencer list, after the selected entry. You can specify <AfterIndex> to select that entry. Or you can select an entry via method RsCmwGprfGen.Source.Sequencer.ListPy.index.

param after_index
integer Index of the entry to be selected

6.3.6.3.4.3 Mdown

SCPI Command :

```
SOURce:GPRF:GENerator<Instance>:SEQuencer:LIST:ENTRy:MDOWn
```

class MdownCls

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

set(*index: int = None*) → None

```
# SCPI: SOURce:GPRF:GENerator<Instance>:SEQuencer:LIST:ENTRy:MDOWn
driver.source.sequencer.listPy.entry.mdown.set(index = 1)
```

Moves the selected entry of the sequencer list one position down. You can specify <Index> to select that entry. Or you can select an entry via method RsCmwGprfGen.Source.Sequencer.ListPy.index. The selection moves with the entry.

param index
integer Index of the entry to be moved

6.3.6.3.4.4 Mup

SCPI Command :

```
SOURce:GPRF:GENerator<Instance>:SEQuencer:LIST:ENTRy:MUP
```

`class MupCls`

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

`set`(*index: int = None*) → None

```
# SCPI: SOURce:GPRF:GENerator<Instance>:SEQuencer:LIST:ENTRy:MUP
driver.source.sequencer.listPy.entry.mup.set(index = 1)
```

Moves the selected entry of the sequencer list one position up. You can specify <Index> to select that entry. Or you can select an entry via method RsCmwGprfGen.Source.Sequencer.ListPy.index. The selection moves with the entry.

param index
integer Index of the entry to be moved

6.3.6.3.5 Fill

SCPI Commands :

```
SOURce:GPRF:GENerator<Instance>:SEQuencer:LIST:FILL:SINDex
SOURce:GPRF:GENerator<Instance>:SEQuencer:LIST:FILL:RANGe
```

`class FillCls`

Fill commands group definition. 12 total commands, 4 Subgroups, 2 group commands

`get_range`() → int

```
# SCPI: SOURce:GPRF:GENerator<Instance>:SEQuencer:LIST:FILL:RANGe
value: int = driver.source.sequencer.listPy.fill.get_range()
```

Specifies the number of entries to be filled. The maximum is limited by 2000 minus the start index of the sequence.

return
range_py: integer Range: 1 to 2000

`get_sindex`() → int

```
# SCPI: SOURce:GPRF:GENerator<Instance>:SEQuencer:LIST:FILL:SINDex
value: int = driver.source.sequencer.listPy.fill.get_sindex()
```

Selects the first index of the sequence to be filled. The maximum value is limited by the index of the highest existing entry plus 1.

return
start_index: integer Range: 0 to 1999

set_range(*range_py: int*) → None

```
# SCPI: SOURce:GPRF:GENerator<Instance>:SEQuencer:LIST:FILL:RANGe
driver.source.sequencer.listPy.fill.set_range(range_py = 1)
```

Specifies the number of entries to be filled. The maximum is limited by 2000 minus the start index of the sequence.

param range_py
integer Range: 1 to 2000

set_sindex(*start_index: int*) → None

```
# SCPI: SOURce:GPRF:GENerator<Instance>:SEQuencer:LIST:FILL:SINDex
driver.source.sequencer.listPy.fill.set_sindex(start_index = 1)
```

Selects the first index of the sequence to be filled. The maximum value is limited by the index of the highest existing entry plus 1.

param start_index
integer Range: 0 to 1999

Cloning the Group

```
# Create a clone of the original group, that exists independently
group2 = driver.source.sequencer.listPy.fill.clone()
```

Subgroups

6.3.6.3.5.1 Apply

SCPI Command :

```
SOURce:GPRF:GENerator<Instance>:SEQuencer:LIST:FILL:APPLy
```

class ApplyCls

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

set() → None

```
# SCPI: SOURce:GPRF:GENerator<Instance>:SEQuencer:LIST:FILL:APPLy
driver.source.sequencer.listPy.fill.apply.set()
```

Fills the sequencer list with a sequence of entries, as configured by the other SOURce:GPRF:GEN<i>:SEQuencer:LIST:FILL:... commands.

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

```
# SCPI: SOURce:GPRF:GENerator<Instance>:SEQuencer:LIST:FILL:APPLy
driver.source.sequencer.listPy.fill.apply.set_with_opc()
```

Fills the sequencer list with a sequence of entries, as configured by the other SOURce:GPRF:GEN<i>:SEQuencer:LIST:FILL:... commands.

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

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

6.3.6.3.5.2 Dgain

SCPI Commands :

```
SOURce:GPRF:GENerator<Instance>:SEQuencer:LIST:FILL:DGAin:SVALue
SOURce:GPRF:GENerator<Instance>:SEQuencer:LIST:FILL:DGAin:INCRement
SOURce:GPRF:GENerator<Instance>:SEQuencer:LIST:FILL:DGAin:KEEP
```

class DgainCls

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

get_increment() → float

```
# SCPI: SOURce:GPRF:GENerator<Instance>:SEQuencer:LIST:FILL:DGAin:INCRement
value: float = driver.source.sequencer.listPy.fill.dgain.get_increment()
```

Configures the increment for filling the sequencer list with digital gain values.

return
increment: numeric Range: Depends on the number of entries and on the start value , Unit: dB

get_keep() → bool

```
# SCPI: SOURce:GPRF:GENerator<Instance>:SEQuencer:LIST:FILL:DGAin:KEEP
value: bool = driver.source.sequencer.listPy.fill.dgain.get_keep()
```

Selects whether the digital gain of existing entries is kept or overwritten when the sequencer list is filled.

return
keep_flag: OFF | ON OFF: overwrite values ON: keep values

get_svalue() → float

```
# SCPI: SOURce:GPRF:GENerator<Instance>:SEQuencer:LIST:FILL:DGAin:SVALue
value: float = driver.source.sequencer.listPy.fill.dgain.get_svalue()
```

Configures the start value for filling the sequencer list with digital gain values.

return
start_value: numeric Unit: dB

set_increment(*increment: float*) → None

```
# SCPI: SOURce:GPRF:GENerator<Instance>:SEQuencer:LIST:FILL:DGAin:INCRement
driver.source.sequencer.listPy.fill.dgain.set_increment(increment = 1.0)
```

Configures the increment for filling the sequencer list with digital gain values.

param increment
numeric Range: Depends on the number of entries and on the start value , Unit: dB

set_keep(*keep_flag: bool*) → None

```
# SCPI: SOURce:GPRF:GENerator<Instance>:SEQuencer:LIST:FILL:DGAin:KEEP
driver.source.sequencer.listPy.fill.dgain.set_keep(keep_flag = False)
```

Selects whether the digital gain of existing entries is kept or overwritten when the sequencer list is filled.

param keep_flag
OFF | ON OFF: overwrite values ON: keep values

set_svalue(*start_value: float*) → None

```
# SCPI: SOURce:GPRF:GENerator<Instance>:SEQuencer:LIST:FILL:DGAin:SVALue
driver.source.sequencer.listPy.fill.dgain.set_svalue(start_value = 1.0)
```

Configures the start value for filling the sequencer list with digital gain values.

param start_value
numeric Unit: dB

6.3.6.3.5.3 Frequency

SCPI Commands :

```
SOURce:GPRF:GENerator<Instance>:SEQuencer:LIST:FILL:FREQuency:SVALue
SOURce:GPRF:GENerator<Instance>:SEQuencer:LIST:FILL:FREQuency:INCRement
SOURce:GPRF:GENerator<Instance>:SEQuencer:LIST:FILL:FREQuency:KEEP
```

class FrequencyCls

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

get_increment() → float

```
# SCPI: SOURce:GPRF:GENerator<Instance>:SEQuencer:LIST:FILL:FREQuency:INCRement
value: float = driver.source.sequencer.listPy.fill.frequency.get_increment()
```

Configures the increment for filling the sequencer list with frequency values.

return
increment: numeric Range: Depends on the number of entries and on the start value , Unit: Hz

get_keep() → bool

```
# SCPI: SOURce:GPRF:GENerator<Instance>:SEQuencer:LIST:FILL:FREQuency:KEEP
value: bool = driver.source.sequencer.listPy.fill.frequency.get_keep()
```

Selects whether the frequency of existing entries is kept or overwritten when the sequencer list is filled.

return
keep_flag: OFF | ON OFF: overwrite values ON: keep values

get_svalue() → float

```
# SCPI: SOURce:GPRF:GENerator<Instance>:SEQuencer:LIST:FILL:FREQuency:SVALue
value: float = driver.source.sequencer.listPy.fill.frequency.get_svalue()
```

Configures the start value for filling the sequencer list with frequency values. For the supported frequency range, see 'Frequency ranges'.

return
start_value: numeric Unit: Hz

set_increment(*increment: float*) → None

```
# SCPI: SOURce:GPRF:GENerator<Instance>:SEQuencer:LIST:FILL:FREQuency:INCRement
driver.source.sequencer.listPy.fill.frequency.set_increment(increment = 1.0)
```

Configures the increment for filling the sequencer list with frequency values.

param increment
numeric Range: Depends on the number of entries and on the start value , Unit: Hz

set_keep(*keep_flag: bool*) → None

```
# SCPI: SOURce:GPRF:GENerator<Instance>:SEQuencer:LIST:FILL:FREQuency:KEEP
driver.source.sequencer.listPy.fill.frequency.set_keep(keep_flag = False)
```

Selects whether the frequency of existing entries is kept or overwritten when the sequencer list is filled.

param keep_flag
OFF | ON OFF: overwrite values ON: keep values

set_svalue(*start_value: float*) → None

```
# SCPI: SOURce:GPRF:GENerator<Instance>:SEQuencer:LIST:FILL:FREQuency:SVALue
driver.source.sequencer.listPy.fill.frequency.set_svalue(start_value = 1.0)
```

Configures the start value for filling the sequencer list with frequency values. For the supported frequency range, see 'Frequency ranges'.

param start_value
numeric Unit: Hz

6.3.6.3.5.4 Lrms

SCPI Commands :

```
SOURce:GPRF:GENerator<Instance>:SEQuencer:LIST:FILL:LRMS:SVALue
SOURce:GPRF:GENerator<Instance>:SEQuencer:LIST:FILL:LRMS:INCRement
SOURce:GPRF:GENerator<Instance>:SEQuencer:LIST:FILL:LRMS:KEEP
```

class LrmsCls

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

get_increment() → float

```
# SCPI: SOURce:GPRF:GENerator<Instance>:SEQuencer:LIST:FILL:LRMS:INCRement
value: float = driver.source.sequencer.listPy.fill.lrms.get_increment()
```

Configures the increment for filling the sequencer list with level values.

return
increment: numeric Range: Depends on the number of entries and on the start value , Unit: dBm

get_keep() → bool

```
# SCPI: SOURce:GPRF:GENerator<Instance>:SEQuencer:LIST:FILL:LRMS:KEEP
value: bool = driver.source.sequencer.listPy.fill.lrms.get_keep()
```

Selects whether the level of existing entries is kept or overwritten when the sequencer list is filled.

return
keep_flag: OFF | ON OFF: overwrite values ON: keep values

get_svalue() → float

```
# SCPI: SOURce:GPRF:GENerator<Instance>:SEQuencer:LIST:FILL:LRMS:SVALue
value: float = driver.source.sequencer.listPy.fill.lrms.get_svalue()
```

Configures the start value for filling the sequencer list with level values.

return
start_value: numeric Range: Please notice the ranges quoted in the data sheet. , Unit: dBm

set_increment(*increment: float*) → None

```
# SCPI: SOURce:GPRF:GENerator<Instance>:SEQuencer:LIST:FILL:LRMS:INCRement
driver.source.sequencer.listPy.fill.lrms.set_increment(increment = 1.0)
```

Configures the increment for filling the sequencer list with level values.

param increment
numeric Range: Depends on the number of entries and on the start value , Unit: dBm

set_keep(*keep_flag: bool*) → None

```
# SCPI: SOURce:GPRF:GENerator<Instance>:SEQuencer:LIST:FILL:LRMS:KEEP
driver.source.sequencer.listPy.fill.lrms.set_keep(keep_flag = False)
```

Selects whether the level of existing entries is kept or overwritten when the sequencer list is filled.

param keep_flag
OFF | ON OFF: overwrite values ON: keep values

set_svalue(*start_value: float*) → None

```
# SCPI: SOURce:GPRF:GENerator<Instance>:SEQuencer:LIST:FILL:LRMS:SVALue
driver.source.sequencer.listPy.fill.lrms.set_svalue(start_value = 1.0)
```

Configures the start value for filling the sequencer list with level values.

param start_value
numeric Range: Please notice the ranges quoted in the data sheet. , Unit: dBm

6.3.6.3.6 Frequency

SCPI Commands :

```
SOURce:GPRF:GENerator<Instance>:SEQuencer:LIST:FREQuency
SOURce:GPRF:GENerator<Instance>:SEQuencer:LIST:FREQuency:ALL
```

`class FrequencyCls`

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

`get`(*index: int*) → float

```
# SCPI: SOURce:GPRF:GENerator<Instance>:SEQuencer:LIST:FREQuency
value: float = driver.source.sequencer.listPy.frequency.get(index = 1)
```

Defines or queries the RF generator frequency for the sequencer list entry with the selected <Index>. For the supported frequency range, see 'Frequency ranges'.

param index
integer

return
frequency: numeric Unit: Hz

`get_all`() → List[float]

```
# SCPI: SOURce:GPRF:GENerator<Instance>:SEQuencer:LIST:FREQuency:ALL
value: List[float] = driver.source.sequencer.listPy.frequency.get_all()
```

Defines the RF generator frequencies for all sequencer list entries. For the supported frequency range, see 'Frequency ranges'.

return
frequency: numeric Comma-separated list of values, one value per list entry Unit: Hz

`set`(*index: int*, *frequency: float*) → None

```
# SCPI: SOURce:GPRF:GENerator<Instance>:SEQuencer:LIST:FREQuency
driver.source.sequencer.listPy.frequency.set(index = 1, frequency = 1.0)
```

Defines or queries the RF generator frequency for the sequencer list entry with the selected <Index>. For the supported frequency range, see 'Frequency ranges'.

param index
integer

param frequency
numeric Unit: Hz

`set_all`(*frequency: List[float]*) → None

```
# SCPI: SOURce:GPRF:GENerator<Instance>:SEQuencer:LIST:FREQuency:ALL
driver.source.sequencer.listPy.frequency.set_all(frequency = [1.1, 2.2, 3.3])
```

Defines the RF generator frequencies for all sequencer list entries. For the supported frequency range, see 'Frequency ranges'.

param frequency
numeric Comma-separated list of values, one value per list entry Unit: Hz

6.3.6.3.7 Itransition

SCPI Commands :

```
SOURce:GPRF:GENerator<Instance>:SEQuencer:LIST:ITRansition
SOURce:GPRF:GENerator<Instance>:SEQuencer:LIST:ITRansition:ALL
```

class ItransitionCls

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

get(*index: int*) → IncTransition

```
# SCPI: SOURce:GPRF:GENerator<Instance>:SEQuencer:LIST:ITRansition
value: enums.IncTransition = driver.source.sequencer.listPy.itransition.
↪get(index = 1)
```

Defines or queries a condition for the transition to the next list entry, for the sequencer list entry with the selected <Index>.

param index
integer

return
inc_transition: IMMediate | RMARker | WMA1 | WMA2 | WMA3 | WMA4 IMMediate: immediately RMARker: restart marker WMA1 to WMA4: waveform markers 1 to 4

get_all() → List[IncTransition]

```
# SCPI: SOURce:GPRF:GENerator<Instance>:SEQuencer:LIST:ITRansition:ALL
value: List[enums.IncTransition] = driver.source.sequencer.listPy.itransition.
↪get_all()
```

Defines or queries a condition for the transition to the next list entry, for all sequencer list entries.

return
inc_transition: IMMediate | RMARker | WMA1 | WMA2 | WMA3 | WMA4 Comma-separated list of values, one value per list entry. Immediate, restart marker, waveform marker 1 to 4.

set(*index: int*, *inc_transition: IncTransition*) → None

```
# SCPI: SOURce:GPRF:GENerator<Instance>:SEQuencer:LIST:ITRansition
driver.source.sequencer.listPy.itransition.set(index = 1, inc_transition =␣
↪enums.IncTransition.IMMediate)
```

Defines or queries a condition for the transition to the next list entry, for the sequencer list entry with the selected <Index>.

param index
integer

param inc_transition
IMMediate | RMARker | WMA1 | WMA2 | WMA3 | WMA4 IMMediate: immediately RMARker: restart marker WMA1 to WMA4: waveform markers 1 to 4

`set_all`(*inc_transition: List[IncTransition]*) → None

```
# SCPI: SOURce:GPRF:GENerator<Instance>:SEQuencer:LIST:ITRansition:ALL
driver.source.sequencer.listPy.itransition.set_all(inc_transition =␣
↪[IncTransition.IMMediate, IncTransition.WMA4])
```

Defines or queries a condition for the transition to the next list entry, for all sequencer list entries.

param inc_transition
IMMediate | RMARker | WMA1 | WMA2 | WMA3 | WMA4 Comma-separated list of values, one value per list entry. Immediate, restart marker, waveform marker 1 to 4.

6.3.6.3.8 Lincrement

SCPI Commands :

```
SOURce:GPRF:GENerator<Instance>:SEQuencer:LIST:LINCrement
SOURce:GPRF:GENerator<Instance>:SEQuencer:LIST:LINCrement:ALL
```

`class LincrementCls`

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

`get`(*index: int*) → ListIncrement

```
# SCPI: SOURce:GPRF:GENerator<Instance>:SEQuencer:LIST:LINCrement
value: enums.ListIncrement = driver.source.sequencer.listPy.lincrement.
↪get(index = 1)
```

Defines or queries the list increment for the sequencer list entry with the selected <Index>.

param index
integer

return
list_increment: DTIMe | ACYCles | USER | MEASurement | TRIGger DTIMe Dwell time defined via method RsCmwGprfGen.Source.Sequencer.ListPy.Dtime.set. ACYCles ARB cycles defined via method RsCmwGprfGen.Source.Sequencer.ListPy.Acycles.set. USER User action triggered via method RsCmwGprfGen.Trigger.Sequencer.Manual.Execute.set. MEASurement Measurement source selected via method RsCmwGprfGen.Trigger.Sequencer.IsMeas.source. TRIGger Trigger source selected via method RsCmwGprfGen.Trigger.Sequencer.IsTrigger.source.

`get_all`() → List[ListIncrement]

```
# SCPI: SOURce:GPRF:GENerator<Instance>:SEQuencer:LIST:LINCrement:ALL
value: List[enums.ListIncrement] = driver.source.sequencer.listPy.lincrement.
↪get_all()
```

Defines the list increments for all sequencer list entries.

return
list_increment: DTIMe | ACYCles | USER | MEASurement | TRIGger Comma-separated list of values, one value per list entry DTIMe Dwell time defined via method RsCmwGprfGen.Source.Sequencer.ListPy.Dtime.all. ACYCles ARB cycles defined via method RsCmwGprfGen.Source.Sequencer.ListPy.Acycles.all. USER User action triggered via method RsCmwGprfGen.Trigger.Sequencer.Manual.Execute.set. MEASurement Measurement source selected via method RsCmwGprfGen.Trigger.Sequencer.IsMeas.source. TRIGger Trigger source selected via method RsCmwGprfGen.Trigger.Sequencer.IsTrigger.source.

set(*index: int*, *list_increment: ListIncrement*) → None

```
# SCPI: SOURce:GPRF:GENerator<Instance>:SEQuencer:LIST:LINCrement
driver.source.sequencer.listPy.lincrement.set(index = 1, list_increment = enums.
↪ListIncrement.ACYCles)
```

Defines or queries the list increment for the sequencer list entry with the selected <Index>.

param index
integer

param list_increment
DTIMe | ACYCles | USER | MEASurement | TRIGger DTIMe Dwell time defined via method RsCmwGprfGen.Source.Sequencer.ListPy.Dtime.set. ACYCles ARB cycles defined via method RsCmwGprfGen.Source.Sequencer.ListPy.Acycles.set. USER User action triggered via method RsCmwGprfGen.Trigger.Sequencer.Manual.Execute.set. MEASurement Measurement source selected via method RsCmwGprfGen.Trigger.Sequencer.IsMeas.source. TRIGger Trigger source selected via method RsCmwGprfGen.Trigger.Sequencer.IsTrigger.source.

set_all(*list_increment: List[ListIncrement]*) → None

```
# SCPI: SOURce:GPRF:GENerator<Instance>:SEQuencer:LIST:LINCrement:ALL
driver.source.sequencer.listPy.lincrement.set_all(list_increment =␣
↪[ListIncrement.ACYCles, ListIncrement.USER])
```

Defines the list increments for all sequencer list entries.

param list_increment
DTIMe | ACYCles | USER | MEASurement | TRIGger Comma-separated list of values, one value per list entry DTIMe Dwell time defined via method RsCmwGprfGen.Source.Sequencer.ListPy.Dtime.all. ACYCles ARB cycles defined via method RsCmwGprfGen.Source.Sequencer.ListPy.Acycles.all. USER User action triggered via method RsCmwGprfGen.Trigger.Sequencer.Manual.Execute.set. MEASurement Measurement source selected via method RsCmwGprfGen.Trigger.Sequencer.IsMeas.source. TRIGger Trigger source selected via method RsCmwGprfGen.Trigger.Sequencer.IsTrigger.source.

6.3.6.3.9 Lrms

SCPI Commands :

```
SOURce:GPRF:GENerator<Instance>:SEQuencer:LIST:LRMS
SOURce:GPRF:GENerator<Instance>:SEQuencer:LIST:LRMS:ALL
```

class LrmsCls

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

get(*index: int*) → float

```
# SCPI: SOURce:GPRF:GENerator<Instance>:SEQuencer:LIST:LRMS
value: float = driver.source.sequencer.listPy.lrms.get(index = 1)
```

Defines or queries the level for the sequencer list entry with the selected <Index>.

param index
integer

return
level_rms: numeric Range: Please notice the ranges quoted in the data sheet. , Unit: dBm

get_all() → List[float]

```
# SCPI: SOURce:GPRF:GENerator<Instance>:SEQuencer:LIST:LRMS:ALL
value: List[float] = driver.source.sequencer.listPy.lrms.get_all()
```

Defines the level for all sequencer list entries.

return
level_rms: numeric Comma-separated list of values, one value per list entry Range: Please notice the ranges quoted in the data sheet. , Unit: dBm

set(*index: int*, *level_rms: float*) → None

```
# SCPI: SOURce:GPRF:GENerator<Instance>:SEQuencer:LIST:LRMS
driver.source.sequencer.listPy.lrms.set(index = 1, level_rms = 1.0)
```

Defines or queries the level for the sequencer list entry with the selected <Index>.

param index
integer

param level_rms
numeric Range: Please notice the ranges quoted in the data sheet. , Unit: dBm

set_all(*level_rms: List[float]*) → None

```
# SCPI: SOURce:GPRF:GENerator<Instance>:SEQuencer:LIST:LRMS:ALL
driver.source.sequencer.listPy.lrms.set_all(level_rms = [1.1, 2.2, 3.3])
```

Defines the level for all sequencer list entries.

param level_rms
numeric Comma-separated list of values, one value per list entry Range: Please notice the ranges quoted in the data sheet. , Unit: dBm

6.3.6.3.10 Signal

SCPI Commands :

```
SOURce:GPRF:GENerator<Instance>:SEQuencer:LIST:SIGNal:CATalog
SOURce:GPRF:GENerator<Instance>:SEQuencer:LIST:SIGNal
SOURce:GPRF:GENerator<Instance>:SEQuencer:LIST:SIGNal:ALL
```

class SignalCls

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

get(*index: int*) → str

```
# SCPI: SOURce:GPRF:GENerator<Instance>:SEQuencer:LIST:SIGNal
value: str = driver.source.sequencer.listPy.signal.get(index = 1)
```

Defines or queries the signal type for the sequencer list entry with the selected <Index>. A complete list of all supported strings can be queried using method RsCmwGprfGen.Source.Sequencer.ListPy.Signal.catalog.

param index
integer

return
signal: string Signal type

get_all() → List[str]

```
# SCPI: SOURce:GPRF:GENerator<Instance>:SEQuencer:LIST:SIGNal:ALL
value: List[str] = driver.source.sequencer.listPy.signal.get_all()
```

Defines the signal types for all sequencer list entries. A complete list of all supported strings can be queried using method RsCmwGprfGen.Source.Sequencer.ListPy.Signal.catalog.

return
signal: string Comma-separated list of strings, one string per list entry

get_catalog() → List[str]

```
# SCPI: SOURce:GPRF:GENerator<Instance>:SEQuencer:LIST:SIGNal:CATalog
value: List[str] = driver.source.sequencer.listPy.signal.get_catalog()
```

Queries all available signal types. The available types depend on the ARB file pool.

return
signal_types: string Comma-separated list of strings, one string per supported signal type

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

```
# SCPI: SOURce:GPRF:GENerator<Instance>:SEQuencer:LIST:SIGNal
driver.source.sequencer.listPy.signal.set(index = 1, signal = 'abc')
```

Defines or queries the signal type for the sequencer list entry with the selected <Index>. A complete list of all supported strings can be queried using method RsCmwGprfGen.Source.Sequencer.ListPy.Signal.catalog.

param index
integer

param signal
string Signal type

`set_all`(*signal: List[str]*) → None

```
# SCPI: SOURce:GPRF:GENerator<Instance>:SEQuencer:LIST:SIGNal:ALL
driver.source.sequencer.listPy.signal.set_all(signal = ['abc1', 'abc2', 'abc3'])
```

Defines the signal types for all sequencer list entries. A complete list of all supported strings can be queried using method RsCmwGprfGen.Source.Sequencer.ListPy.Signal.catalog.

param signal
string Comma-separated list of strings, one string per list entry

6.3.6.3.11 SingleCmw

`class SingleCmwCls`

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

Cloning the Group

```
# Create a clone of the original group, that exists independently
group2 = driver.source.sequencer.listPy.singleCmw.clone()
```

Subgroups

6.3.6.3.11.1 Usage

`class UsageCls`

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

Cloning the Group

```
# Create a clone of the original group, that exists independently
group2 = driver.source.sequencer.listPy.singleCmw.usage.clone()
```

Subgroups

6.3.6.3.11.2 Tx

SCPI Command :

```
SOURce:GPRF:GENerator<Instance>:SEQuencer:LIST:CMWS:USAGe:TX
```

class TxCls

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

get(*index: float*) → List[bool]

```
# SCPI: SOURce:GPRF:GENerator<Instance>:SEQuencer:LIST:CMWS:USAGe:TX
value: List[bool] = driver.source.sequencer.listPy.singleCmw.usage.tx.get(index
↪= 1.0)
```

No command help available

param index
No help available

return
usage: No help available

set(*index: float*, *usage: List[bool]*) → None

```
# SCPI: SOURce:GPRF:GENerator<Instance>:SEQuencer:LIST:CMWS:USAGe:TX
driver.source.sequencer.listPy.singleCmw.usage.tx.set(index = 1.0, usage =
↪[True, False, True])
```

No command help available

param index
No help available

param usage
No help available

6.3.6.3.12 Spath

class SpathCls

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

Cloning the Group

```
# Create a clone of the original group, that exists independently
group2 = driver.source.sequencer.listPy.spath.clone()
```

Subgroups

6.3.6.3.12.1 Usage

SCPI Command :

```
SOURce:GPRF:GENerator<Instance>:SEQuencer:LIST:SPATh:USAGe
```

class UsageCls

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

get(*index: float*) → List[bool]

```
# SCPI: SOURce:GPRF:GENerator<Instance>:SEQuencer:LIST:SPATh:USAGe
value: List[bool] = driver.source.sequencer.listPy.spath.usage.get(index = 1.0)
```

No command help available

param index
No help available

return
enable: No help available

set(*index: float*, *enable: List[bool]*) → None

```
# SCPI: SOURce:GPRF:GENerator<Instance>:SEQuencer:LIST:SPATh:USAGe
driver.source.sequencer.listPy.spath.usage.set(index = 1.0, enable = [True,␣
↪False, True])
```

No command help available

param index
No help available

param enable
No help available

Cloning the Group

```
# Create a clone of the original group, that exists independently
group2 = driver.source.sequencer.listPy.spath.usage.clone()
```

Subgroups

6.3.6.3.12.2 Bench<Bench>

RepCap Settings

```
# Range: Nr1 .. Nr20
rc = driver.source.sequencer.listPy.spath.usage.bench.repcap_bench_get()
driver.source.sequencer.listPy.spath.usage.bench.repcap_bench_set(repcap.Bench.Nr1)
```

SCPI Command :

```
SOURce:GPRF:GENerator<Instance>:SEQuencer:LIST:SPATh:USAGe:BENCh<nr>
```

class BenchCls

Bench commands group definition. 1 total commands, 0 Subgroups, 1 group commands Repeated Capability: Bench, default value after init: Bench.Nr1

get(*index: float*, *bench=Bench.Default*) → List[bool]

```
# SCPI: SOURce:GPRF:GENerator<Instance>:SEQuencer:LIST:SPATh:USAGe:BENCh<nr>
value: List[bool] = driver.source.sequencer.listPy.spath.usage.bench.get(index
↪= 1.0, bench = repcap.Bench.Default)
```

No command help available

param index
No help available

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

return
enable: No help available

set(*index: float*, *enable: List[bool]*, *bench=Bench.Default*) → None

```
# SCPI: SOURce:GPRF:GENerator<Instance>:SEQuencer:LIST:SPATh:USAGe:BENCh<nr>
driver.source.sequencer.listPy.spath.usage.bench.set(index = 1.0, enable =
↪[True, False, True], bench = repcap.Bench.Default)
```

No command help available

param index
No help available

param enable
No help available

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

Cloning the Group

```
# Create a clone of the original group, that exists independently
group2 = driver.source.sequencer.listPy.spath.usage.bench.clone()
```

6.3.6.3.13 SymbolRate

SCPI Commands :

```
SOURce:GPRF:GENerator<Instance>:SEQuencer:LIST:SRATe
SOURce:GPRF:GENerator<Instance>:SEQuencer:LIST:SRATe:ALL
```

class SymbolRateCls

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

get(*index: int*) → float

```
# SCPI: SOURce:GPRF:GENerator<Instance>:SEQuencer:LIST:SRATe
value: float = driver.source.sequencer.listPy.symbolRate.get(index = 1)
```

Queries the sample rate for the sequencer list entry with the selected <Index>.

param index
integer

return
sample_rate: float Unit: Samples per second

`get_all()` → List[float]

```
# SCPI: SOURce:GPRF:GENerator<Instance>:SEQuencer:LIST:SRATe:ALL
value: List[float] = driver.source.sequencer.listPy.symbolRate.get_all()
```

Queries the sample rates for all sequencer list entries.

return
sample_rate: float Comma-separated list of values, one value per list entry Unit: Samples per second

6.3.6.3.14 Ttime

SCPI Commands :

```
SOURce:GPRF:GENerator<Instance>:SEQuencer:LIST:TTIMe
SOURce:GPRF:GENerator<Instance>:SEQuencer:LIST:TTIMe:ALL
```

`class TtimeCls`

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

`get`(*index: int*) → float

```
# SCPI: SOURce:GPRF:GENerator<Instance>:SEQuencer:LIST:TTIMe
value: float = driver.source.sequencer.listPy.ttime.get(index = 1)
```

Queries the transition time for the sequencer list entry with the selected <Index>.

param index
integer

return
trans_time: float Range: 0 s to 500E-6 s, Unit: s

`get_all()` → List[float]

```
# SCPI: SOURce:GPRF:GENerator<Instance>:SEQuencer:LIST:TTIMe:ALL
value: List[float] = driver.source.sequencer.listPy.ttime.get_all()
```

Queries the transition times for all sequencer list entries.

return
trans_time: float Comma-separated list of values, one value per list entry Range: 0 s to 500E-6 s, Unit: s

6.3.6.4 Marker

class MarkerCls

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

Cloning the Group

```
# Create a clone of the original group, that exists independently
group2 = driver.source.sequencer.marker.clone()
```

Subgroups

6.3.6.4.1 Delays

SCPI Command :

```
SOURce:GPRF:GENerator<Instance>:SEQuencer:MARKer:DELays
```

class DelaysCls

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

class DelaysStruct

Response structure. Fields:

- Restart_Marker: float: numeric Range: 0 s to 0.1 s, Unit: s
- Marker_2: float: numeric Range: 0 s to 0.1 s, Unit: s
- Marker_3: float: numeric Range: 0 s to 0.1 s, Unit: s
- Marker_4: float: numeric Range: 0 s to 0.1 s, Unit: s

get() → DelaysStruct

```
# SCPI: SOURce:GPRF:GENerator<Instance>:SEQuencer:MARKer:DELays
value: DelaysStruct = driver.source.sequencer.marker.delays.get()
```

Defines delay times for the ARB output trigger events relative to the marker events.

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

set(*restart_marker: float*, *marker_2: float*, *marker_3: float*, *marker_4: float*) → None

```
# SCPI: SOURce:GPRF:GENerator<Instance>:SEQuencer:MARKer:DELays
driver.source.sequencer.marker.delays.set(restart_marker = 1.0, marker_2 = 1.0,␣
↪marker_3 = 1.0, marker_4 = 1.0)
```

Defines delay times for the ARB output trigger events relative to the marker events.

param restart_marker
numeric Range: 0 s to 0.1 s, Unit: s

param marker_2
numeric Range: 0 s to 0.1 s, Unit: s

param marker_3
numeric Range: 0 s to 0.1 s, Unit: s

param marker_4
numeric Range: 0 s to 0.1 s, Unit: s

Cloning the Group

```
# Create a clone of the original group, that exists independently
group2 = driver.source.sequencer.marker.delays.clone()
```

Subgroups

6.3.6.4.1.1 All

SCPI Command :

```
SOURce:GPRF:GENerator<Instance>:SEQuencer:MARKer:DELays:ALL
```

class AllCls

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

class AllStruct

Response structure. Fields:

- Restart_Marker: float: No parameter help available
- Marker_1: float: No parameter help available
- Marker_2: float: No parameter help available
- Marker_3: float: No parameter help available
- Marker_4: float: No parameter help available

get() → AllStruct

```
# SCPI: SOURce:GPRF:GENerator<Instance>:SEQuencer:MARKer:DELays:ALL
value: AllStruct = driver.source.sequencer.marker.delays.all.get()
```

No command help available

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

set(*restart_marker: float*, *marker_1: float*, *marker_2: float*, *marker_3: float*, *marker_4: float*) → None

```
# SCPI: SOURce:GPRF:GENerator<Instance>:SEQuencer:MARKer:DELays:ALL
driver.source.sequencer.marker.delays.all.set(restart_marker = 1.0, marker_1 =␣
↪1.0, marker_2 = 1.0, marker_3 = 1.0, marker_4 = 1.0)
```

No command help available

param restart_marker
No help available

param marker_1
No help available

param marker_2
No help available

param marker_3
No help available

param marker_4
No help available

6.3.6.5 RfSettings

class RfSettingsCls

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

Cloning the Group

```
# Create a clone of the original group, that exists independently
group2 = driver.source.sequencer.rfSettings.clone()
```

Subgroups

6.3.6.5.1 SingleCmw

SCPI Command :

```
SOURce:GPRF:GENerator<Instance>:SEQuencer:RFSettings:CMWS:CSET
```

class SingleCmwCls

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

get_cset() → ParameterSetMode

```
# SCPI: SOURce:GPRF:GENerator<Instance>:SEQuencer:RFSettings:CMWS:CSET
value: enums.ParameterSetMode = driver.source.sequencer.rfSettings.singleCmw.
↪get_cset()
```

No command help available

return
cmws_connector_set: No help available

set_cset(*cmws_connector_set: ParameterSetMode*) → None

```
# SCPI: SOURce:GPRF:GENerator<Instance>:SEQuencer:RFSettings:CMWS:CSET
driver.source.sequencer.rfSettings.singleCmw.set_cset(cmws_connector_set =␣
↪enums.ParameterSetMode.GLOBal)
```

No command help available

param cmws_connector_set
No help available

6.3.6.5.2 Spath

SCPI Command :

```
SOURce:GPRF:GENerator<Instance>:SEQuencer:RFSettings:SPATh:CSET
```

class SpathCls

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

get_cset() → ParameterSetMode

```
# SCPI: SOURce:GPRF:GENerator<Instance>:SEQuencer:RFSettings:SPATh:CSET
value: enums.ParameterSetMode = driver.source.sequencer.rfSettings.spath.get_
↪cset()
```

No command help available

return
connector_set: No help available

set_cset(*connector_set: ParameterSetMode*) → None

```
# SCPI: SOURce:GPRF:GENerator<Instance>:SEQuencer:RFSettings:SPATh:CSET
driver.source.sequencer.rfSettings.spath.set_cset(connector_set = enums.
↪ParameterSetMode.GLOBal)
```

No command help available

param connector_set
No help available

6.3.6.6 Rmarker

SCPI Command :

```
SOURce:GPRF:GENerator<Instance>:SEQuencer:RMARker:DELay
```

class RmarkerCls

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

get_delay() → float

```
# SCPI: SOURce:GPRF:GENerator<Instance>:SEQuencer:RMARker:DELay
value: float = driver.source.sequencer.rmarker.get_delay()
```

Defines a delay time for the ARB output trigger events relative to the restart marker events.

return
restart_marker: numeric Range: 0 s to 0.1 s, Unit: s

set_delay(*restart_marker: float*) → None

```
# SCPI: SOURce:GPRF:GENerator<Instance>:SEQuencer:RMARker:DELay
driver.source.sequencer.rmarker.set_delay(restart_marker = 1.0)
```

Defines a delay time for the ARB output trigger events relative to the restart marker events.

param restart_marker
numeric Range: 0 s to 0.1 s, Unit: s

6.3.6.7 State

SCPI Command :

```
SOURce:GPRF:GENerator<Instance>:SEQuencer:STATe
```

class StateCls

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

get() → GeneratorState

```
# SCPI: SOURce:GPRF:GENerator<Instance>:SEQuencer:STATe
value: enums.GeneratorState = driver.source.sequencer.state.get()
```

Turns the generator on or off.

return
generator_state: OFF | PENDing | ON | RDY OFF: generator switched off PEND: state transition ongoing ON: generator switched on, signal available RDY: generator switched off, sequencer list processing complete for 'Repetition'='Single'

set(*control: bool*) → None

```
# SCPI: SOURce:GPRF:GENerator<Instance>:SEQuencer:STATe
driver.source.sequencer.state.set(control = False)
```

Turns the generator on or off.

param control
ON | OFF Switch the generator ON or OFF.

Cloning the Group

```
# Create a clone of the original group, that exists independently
group2 = driver.source.sequencer.state.clone()
```

Subgroups

6.3.6.7.1 All

SCPI Command :

```
SOURce:GPRF:GENerator<Instance>:SEQuencer:STATe:ALL
```

class AllCls

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

get(*timeout: float = None*, *target_main_state: TargetMainState = None*, *target_sync_state: TargetSyncState = None*) → List[GeneratorState]

```
# SCPI: SOURce:GPRF:GENerator<Instance>:SEQuencer:STATe:ALL
value: List[enums.GeneratorState] = driver.source.sequencer.state.all.
↪get(timeout = 1.0, target_main_state = enums.TargetMainState.OFF, target_sync_
↪state = enums.TargetSyncState.ADJusted)
```

Queries the states of the generator. Without query parameters, the states are returned immediately. With query parameters, the states are returned when the <TargetMainState> and the <TargetSyncState> are reached or when the <Timeout> expires.

param timeout
numeric Unit: ms

param target_main_state
OFF | RUN | STOP Target MainState for the query OFF: main state OFF RUN: main state ON STOP: main state RDY Default is RUN.

param target_sync_state
PENDing | ADJusted Target SyncState for the query Default is ADJ.

return
all_states: No help available

6.3.6.8 Tdd

SCPI Command :

```
SOURce:GPRF:GENerator<Instance>:SEQuencer:TDD:MODE
```

class TddCls

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

get_mode() → bool

```
# SCPI: SOURce:GPRF:GENerator<Instance>:SEQuencer:TDD:MODE
value: bool = driver.source.sequencer.tdd.get_mode()
```

No command help available

return
tdd_mode: No help available

set_mode(*tdd_mode: bool*) → None

```
# SCPI: SOURce:GPRF:GENerator<Instance>:SEQuencer:TDD:MODE
driver.source.sequencer.tdd.set_mode(tdd_mode = False)
```

No command help available

param tdd_mode
No help available

6.3.6.9 Wmarker<Marker>

RepCap Settings

```
# Range: Nr1 .. Nr4
rc = driver.source.sequencer.wmarker.repcap_marker_get()
driver.source.sequencer.wmarker.repcap_marker_set(repcap.Marker.Nr1)
```

class WmarkerCls

Wmarker commands group definition. 2 total commands, 1 Subgroups, 0 group commands Repeated Capability: Marker, default value after init: Marker.Nr1

Cloning the Group

```
# Create a clone of the original group, that exists independently
group2 = driver.source.sequencer.wmarker.clone()
```

Subgroups

6.3.6.9.1 Delay

SCPI Command :

```
SOURce:GPRF:GENerator<Instance>:SEQuencer:WMARker<no>:DELay
```

class DelayCls

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

get(*marker=Marker.Default*) → float

```
# SCPI: SOURce:GPRF:GENerator<Instance>:SEQuencer:WMARker<no>:DELay
value: float = driver.source.sequencer.wmarker.delay.get(marker = repcap.Marker.
↪Default)
```

Defines a delay time for the ARB output trigger events relative to the waveform marker <no> events.

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

return
waveform_marker: numeric Range: 0 s to 0.1 s, Unit: s

set(*waveform_marker: float*, *marker=Marker.Default*) → None

```
# SCPI: SOURce:GPRF:GENerator<Instance>:SEQuencer:WMARker<no>:DELay
driver.source.sequencer.wmarker.delay.set(waveform_marker = 1.0, marker =␣
↪repcap.Marker.Default)
```

Defines a delay time for the ARB output trigger events relative to the waveform marker <no> events.

param waveform_marker
numeric Range: 0 s to 0.1 s, Unit: s

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

Cloning the Group

```
# Create a clone of the original group, that exists independently
group2 = driver.source.sequencer.wmarker.delay.clone()
```

Subgroups

6.3.6.9.1.1 All

SCPI Command :

```
SOURce:GPRF:GENerator<Instance>:SEQuencer:WMARker:DELay:ALL
```

class AllCls

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

class AllStruct

Response structure. Fields:

- Marker_1: float: No parameter help available
- Marker_2: float: No parameter help available
- Marker_3: float: No parameter help available
- Marker_4: float: No parameter help available

get() → AllStruct

```
# SCPI: SOURce:GPRF:GENerator<Instance>:SEQuencer:WMARker:DELay:ALL
value: AllStruct = driver.source.sequencer.wmarker.delay.all.get()
```

No command help available

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

set(*marker_1: float*, *marker_2: float*, *marker_3: float*, *marker_4: float*) → None

```
# SCPI: SOURce:GPRF:GENerator<Instance>:SEQuencer:WMARker:DELay:ALL
driver.source.sequencer.wmarker.delay.all.set(marker_1 = 1.0, marker_2 = 1.0,␣
↪marker_3 = 1.0, marker_4 = 1.0)
```

No command help available

param marker_1
No help available

param marker_2
No help available

param marker_3
No help available

param marker_4
No help available

6.3.7 State

SCPI Command :

```
SOURce:GPRF:GENerator<Instance>:STATe
```

class StateCls

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

get() → GeneratorState

```
# SCPI: SOURce:GPRF:GENerator<Instance>:STATe
value: enums.GeneratorState = driver.source.state.get()
```

Turns the generator on or off.

return
generator_state: OFF | PENDing | ON | RDY OFF: generator switched off PEND: state transition ongoing ON: generator switched on, signal available RDY: generator switched off, ARB file processing complete in smart channel mode

set(*control: bool*) → None

```
# SCPI: SOURce:GPRF:GENerator<Instance>:STATe
driver.source.state.set(control = False)
```

Turns the generator on or off.

param control
ON | OFF Switch the generator ON or OFF.

Cloning the Group

```
# Create a clone of the original group, that exists independently
group2 = driver.source.state.clone()
```

Subgroups

6.3.7.1 All

SCPI Command :

```
SOURce:GPRF:GENerator<Instance>:STATe:ALL
```

class AllCls

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

get(*timeout: float = None*, *target_main_state: TargetMainState = None*, *target_sync_state: TargetSyncState = None*) → List[GeneratorState]

```
# SCPI: SOURce:GPRF:GENerator<Instance>:STATe:ALL
value: List[enums.GeneratorState] = driver.source.state.all.get(timeout = 1.0,␣
↪target_main_state = enums.TargetMainState.OFF, target_sync_state = enums.
↪TargetSyncState.ADJusted)
```

Queries the states of the generator. Without query parameters, the states are returned immediately. With query parameters, the states are returned when the <TargetMainState> and the <TargetSyncState> are reached or when the <Timeout> expires.

param timeout
numeric Unit: ms

param target_main_state
OFF | RUN | STOP Target MainState for the query OFF: main state OFF RUN: main state ON STOP: main state RDY Default is RUN.

param target_sync_state
PENDing | ADJusted Target SyncState for the query Default is ADJ.

return
all_states: No help available

6.4 Trigger

class TriggerCls

Trigger commands group definition. 15 total commands, 2 Subgroups, 0 group commands

Cloning the Group

```
# Create a clone of the original group, that exists independently
group2 = driver.trigger.clone()
```

Subgroups

6.4.1 Arb

SCPI Commands :

```
TRIGger:GPRF:GENerator<Instance>:ARB:SOURce
TRIGger:GPRF:GENerator<Instance>:ARB:DELay
TRIGger:GPRF:GENerator<Instance>:ARB:SLOPe
TRIGger:GPRF:GENerator<Instance>:ARB:RETRigger
TRIGger:GPRF:GENerator<Instance>:ARB:AUTostart
```

class ArbCls

Arb commands group definition. 9 total commands, 3 Subgroups, 5 group commands

get_autostart() → bool

```
# SCPI: TRIGger:GPRF:GENerator<Instance>:ARB:AUTostart
value: bool = driver.trigger.arb.get_autostart()
```

Enables or disables the automatic download of the selected (and loaded) waveform file whenever the generator is turned on.

return
autostart: OFF | ON Autostart is disabled or enabled.

get_delay() → float

```
# SCPI: TRIGger:GPRF:GENerator<Instance>:ARB:DELay
value: float = driver.trigger.arb.get_delay()
```

Sets the trigger delay. The delay does not apply to manual execution (see method RsCmwGprfGen.Trigger.Arb.Segments.Manual. Execute.set) .

return
delay: numeric Range: 0 s to 100 s, Unit: s

get_retrigger() → bool

```
# SCPI: TRIGger:GPRF:GENerator<Instance>:ARB:RETRigger
value: bool = driver.trigger.arb.get_retrigger()
```

Enables or disables the trigger system for waveform files.

return
retrigger: OFF | ON Trigger system disabled or enabled

get_slope() → SignalSlope

```
# SCPI: TRIGger:GPRF:GENerator<Instance>:ARB:SLOPe
value: enums.SignalSlope = driver.trigger.arb.get_slope()
```

Qualifies whether the trigger event is generated at the rising or at the falling edge of the trigger pulse.

return
slope: REDGe | FEDGe REDGe: Rising edge FEDGe: Falling edge

get_source() → str

```
# SCPI: TRIGger:GPRF:GENerator<Instance>:ARB:SOURce
value: str = driver.trigger.arb.get_source()
```

Selects the source of the trigger events. The supported values depend on the installed options. You can query a list of all supported values via method RsCmwGprfGen.Trigger.Arb.Catalog.source.

return
source: string

set_autostart(*autostart: bool*) → None

```
# SCPI: TRIGger:GPRF:GENerator<Instance>:ARB:AUTostart
driver.trigger.arb.set_autostart(autostart = False)
```

Enables or disables the automatic download of the selected (and loaded) waveform file whenever the generator is turned on.

param autostart
OFF | ON Autostart is disabled or enabled.

set_delay(*delay: float*) → None

```
# SCPI: TRIGger:GPRF:GENerator<Instance>:ARB:DELay
driver.trigger.arb.set_delay(delay = 1.0)
```

Sets the trigger delay. The delay does not apply to manual execution (see method RsCmwGprfGen.Trigger.Arb.Segments.Manual. Execute.set) .

param delay
numeric Range: 0 s to 100 s, Unit: s

set_retrigger(*retrigger: bool*) → None

```
# SCPI: TRIGger:GPRF:GENerator<Instance>:ARB:RETRigger
driver.trigger.arb.set_retrigger(retrigger = False)
```

Enables or disables the trigger system for waveform files.

param retrigger
OFF | ON Trigger system disabled or enabled

set_slope(*slope: SignalSlope*) → None

```
# SCPI: TRIGger:GPRF:GENerator<Instance>:ARB:SLOPe
driver.trigger.arb.set_slope(slope = enums.SignalSlope.FEDGe)
```

Qualifies whether the trigger event is generated at the rising or at the falling edge of the trigger pulse.

param slope
REDGe | FEDGe REDGe: Rising edge FEDGe: Falling edge

set_source(*source: str*) → None

```
# SCPI: TRIGger:GPRF:GENerator<Instance>:ARB:SOURce
driver.trigger.arb.set_source(source = 'abc')
```

Selects the source of the trigger events. The supported values depend on the installed options. You can query a list of all supported values via method RsCmwGprfGen.Trigger.Arb.Catalog.source.

param source
string

Cloning the Group

```
# Create a clone of the original group, that exists independently
group2 = driver.trigger.arb.clone()
```

Subgroups

6.4.1.1 Catalog

SCPI Command :

```
TRIGger:GPRF:GENerator<Instance>:ARB:CATalog:SOURce
```

class CatalogCls

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

get_source() → List[str]

```
# SCPI: TRIGger:GPRF:GENerator<Instance>:ARB:CATalog:SOURce
value: List[str] = driver.trigger.arb.catalog.get_source()
```

Lists all trigger source values that can be set using method RsCmwGprfGen.Trigger.Arb.source.

return

trigger_sources: string Comma-separated list of all supported values. Each value is represented as a string.

6.4.1.2 Manual

class ManualCls

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

Cloning the Group

```
# Create a clone of the original group, that exists independently
group2 = driver.trigger.arb.manual.clone()
```

Subgroups

6.4.1.2.1 Execute

SCPI Command :

```
TRIGger:GPRF:GENerator<Instance>:ARB:MANual:EXECute
```

class ExecuteCls

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

set() → None

```
# SCPI: TRIGger:GPRF:GENerator<Instance>:ARB:MANual:EXECute
driver.trigger.arb.manual.execute.set()
```

Generates a trigger event for the trigger source ‘Manual’.

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

```
# SCPI: TRIGger:GPRF:GENerator<Instance>:ARB:MANual:EXECute
driver.trigger.arb.manual.execute.set_with_opc()
```

Generates a trigger event for the trigger source 'Manual'.

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

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

6.4.1.3 Segments

SCPI Command :

```
TRIGger:GPRF:GENerator<Instance>:ARB:SEGMents:MODE
```

class SegmentsCls

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

get_mode() → ArbSegmentsMode

```
# SCPI: TRIGger:GPRF:GENerator<Instance>:ARB:SEGMents:MODE
value: enums.ArbSegmentsMode = driver.trigger.arb.segments.get_mode()
```

Selects a trigger mode for multi-segment waveform files.

return
mode: CONTinuous | CSEamless | AUTO CONTinuous: A trigger event causes an immediate switchover to the next segment. CSEamless: A trigger event causes a switchover after the end of the segment has been reached. AUTO: The generator processes one segment after another.

set_mode(*mode: ArbSegmentsMode*) → None

```
# SCPI: TRIGger:GPRF:GENerator<Instance>:ARB:SEGMents:MODE
driver.trigger.arb.segments.set_mode(mode = enums.ArbSegmentsMode.AUTO)
```

Selects a trigger mode for multi-segment waveform files.

param mode
CONTinuous | CSEamless | AUTO CONTinuous: A trigger event causes an immediate switchover to the next segment. CSEamless: A trigger event causes a switchover after the end of the segment has been reached. AUTO: The generator processes one segment after another.

Cloning the Group

```
# Create a clone of the original group, that exists independently
group2 = driver.trigger.arb.segments.clone()
```

Subgroups

6.4.1.3.1 Manual

class ManualCls

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

Cloning the Group

```
# Create a clone of the original group, that exists independently
group2 = driver.trigger.arb.segments.manual.clone()
```

Subgroups

6.4.1.3.1.1 Execute

SCPI Command :

```
TRIGger:GPRF:GENerator<Instance>:ARB:SEGMents:MANual:EXECute
```

class ExecuteCls

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

set() → None

```
# SCPI: TRIGger:GPRF:GENerator<Instance>:ARB:SEGMents:MANual:EXECute
driver.trigger.arb.segments.manual.execute.set()
```

Generates a trigger event for the ARB segment trigger. The segment trigger causes the generator to step to the beginning of the next segment in the multi-segment file.

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

```
# SCPI: TRIGger:GPRF:GENerator<Instance>:ARB:SEGMents:MANual:EXECute
driver.trigger.arb.segments.manual.execute.set_with_opc()
```

Generates a trigger event for the ARB segment trigger. The segment trigger causes the generator to step to the beginning of the next segment in the multi-segment file.

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

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

6.4.2 Sequencer

SCPI Command :

```
TRIGger:GPRF:GENerator<Instance>:SEQuencer:TOUT
```

class SequencerCls

Sequencer commands group definition. 6 total commands, 3 Subgroups, 1 group commands

get_timeout() → float

```
# SCPI: TRIGger:GPRF:GENerator<Instance>:SEQuencer:TOUT
value: float or bool = driver.trigger.sequencer.get_timeout()
```

Sets a timeout for waiting for a trigger event for 'List Increment' = 'MEASUREMENT' and 'TRIGGER'.

return
timeout: (float or boolean) float | ON | OFF Range: 0.01 s to 300 s, Unit: s ON | OFF enables or disables the timeout check.

set_timeout(*timeout: float*) → None

```
# SCPI: TRIGger:GPRF:GENerator<Instance>:SEQuencer:TOUT
driver.trigger.sequencer.set_timeout(timeout = 1.0)
```

Sets a timeout for waiting for a trigger event for 'List Increment' = 'MEASUREMENT' and 'TRIGGER'.

param timeout
(float or boolean) float | ON | OFF Range: 0.01 s to 300 s, Unit: s ON | OFF enables or disables the timeout check.

Cloning the Group

```
# Create a clone of the original group, that exists independently
group2 = driver.trigger.sequencer.clone()
```

Subgroups

6.4.2.1 IsMeas

SCPI Commands :

```
TRIGger:GPRF:GENerator<Instance>:SEQuencer:ISMeas:CATalog
TRIGger:GPRF:GENerator<Instance>:SEQuencer:ISMeas:SOURce
```

class IsMeasCls

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

get_catalog() → List[str]

```
# SCPI: TRIGger:GPRF:GENerator<Instance>:SEQuencer:ISMeas:CATalog
value: List[str] = driver.trigger.sequencer.isMeas.get_catalog()
```

Queries all available measurement source strings.

return
source_list: string Comma-separated list of strings. Each string represents a supported source.

get_source() → str

```
# SCPI: TRIGger:GPRF:GENerator<Instance>:SEQuencer:ISMeas:SOURce
value: str = driver.trigger.sequencer.isMeas.get_source()
```

Selects a measurement for triggering sequencer list incrementations. A complete list of all supported strings can be queried using method RsCmwGprfGen.Trigger.Sequencer.IsMeas.catalog.

return
source: string

set_source(*source: str*) → None

```
# SCPI: TRIGger:GPRF:GENerator<Instance>:SEQuencer:ISMeas:SOURce
driver.trigger.sequencer.isMeas.set_source(source = 'abc')
```

Selects a measurement for triggering sequencer list incrementations. A complete list of all supported strings can be queried using method RsCmwGprfGen.Trigger.Sequencer.IsMeas.catalog.

param source
string

6.4.2.2 IsTrigger

SCPI Commands :

```
TRIGger:GPRF:GENerator<Instance>:SEQuencer:ISTRigger:CATalog
TRIGger:GPRF:GENerator<Instance>:SEQuencer:ISTRigger:SOURce
```

class IsTriggerCls

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

get_catalog() → List[str]

```
# SCPI: TRIGger:GPRF:GENerator<Instance>:SEQuencer:ISTRigger:CATalog
value: List[str] = driver.trigger.sequencer.isTrigger.get_catalog()
```

Queries all available trigger source strings.

return
source_list: string Comma-separated list of strings. Each string represents a supported source.

get_source() → str

```
# SCPI: TRIGger:GPRF:GENerator<Instance>:SEQuencer:ISTRigger:SOURce
value: str = driver.trigger.sequencer.isTrigger.get_source()
```

Selects a trigger source for triggering sequencer list incrementations. A complete list of all supported strings can be queried using method RsCmwGprfGen.Trigger.Sequencer.IsTrigger.catalog.

return
source: string

set_source(*source: str*) → None

```
# SCPI: TRIGger:GPRF:GENerator<Instance>:SEQuencer:ISTRigger:SOURce
driver.trigger.sequencer.isTrigger.set_source(source = 'abc')
```

Selects a trigger source for triggering sequencer list incrementations. A complete list of all supported strings can be queried using method RsCmwGprfGen.Trigger.Sequencer.IsTrigger.catalog.

param source
string

6.4.2.3 Manual

class ManualCls

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

Cloning the Group

```
# Create a clone of the original group, that exists independently
group2 = driver.trigger.sequencer.manual.clone()
```

Subgroups

6.4.2.3.1 Execute

SCPI Command :

```
TRIGger:GPRF:GENerator<Instance>:SEQuencer:MANual:EXECute
```

class ExecuteCls

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

set() → None

```
# SCPI: TRIGger:GPRF:GENerator<Instance>:SEQuencer:MANual:EXECute
driver.trigger.sequencer.manual.execute.set()
```

Triggers the transition to the next sequencer list entry manually.

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

```
# SCPI: TRIGger:GPRF:GENerator<Instance>:SEQuencer:MANual:EXECute
driver.trigger.sequencer.manual.execute.set_with_opc()
```

Triggers the transition to the next sequencer list entry manually.

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

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

CHAPTER

SEVEN

RSCMWGPRFGEN UTILITIES

class Utilities

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

Access snippet: `utils = RsCmwGprfGen.utilities`

property logger: *ScpiLogger*

Scpi Logger interface, see *here*

Access snippet: `logger = RsCmwGprfGen.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

go_to_local() → None

Puts the instrument into local state.

go_to_remote() → None

Puts the instrument into remote state.

get_lock() → RLock

Returns the thread lock for the current session.

By default:

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

RSCMWGPRFGEN LOGGER

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

class ScpiLogger

Base class for SCPI logging

mode

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

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

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

RSCMWGPRFGEN 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

U