PyVISA Documentation

Release 1.11.3

PyVISA Authors

Nov 08, 2020

Contents

PyVISA is a Python package that enables you to control all kinds of measurement devices independently of the interface (e.g. GPIB, RS232, USB, Ethernet). As an example, reading self-identification from a Keithley Multimeter with GPIB number 12 is as easy as three lines of Python code:

```
>>> import pyvisa
>>> rm = pyvisa.ResourceManager()
>>> rm.list_resources()
('ASRL1::INSTR', 'ASRL2::INSTR', 'GPIB0::12::INSTR')
>>> inst = rm.open_resource('GPIB0::12::INSTR')
>>> print(inst.query("*IDN?"))
```

(That's the whole program; really!) It works on Windows, Linux and Mac; with arbitrary adapters (e.g. National Instruments, Agilent, Tektronix, Stanford Research Systems).

CHAPTER 1

General overview

The programming of measurement instruments can be real pain. There are many different protocols, sent over many different interfaces and bus systems (e.g. GPIB, RS232, USB, Ethernet). For every programming language you want to use, you have to find libraries that support both your device and its bus system.

In order to ease this unfortunate situation, the Virtual Instrument Software Architecture (VISA) specification was defined in the middle of the 90ies. VISA is a standard for configuring, programming, and troubleshooting instrumentation systems comprising GPIB, VXI, PXI, Serial, Ethernet, and/or USB interfaces.

Today VISA is implemented on all significant operating systems. A couple of vendors offer VISA libraries, partly with free download. These libraries work together with arbitrary peripherical devices, although they may be limited to certain interface devices, such as the vendor's GPIB card.

The VISA specification has explicit bindings to Visual Basic, C, and G (LabVIEW's graphical language). Python can be used to call functions from a VISA shared library (*.dll*, *.so*, *.dylib*) allowing to directly leverage the standard implementations. In addition, Python can be used to directly access most bus systems used by instruments which is why one can envision to implement the VISA standard directly in Python (see the *PyVISA-Py* project for more details). PyVISA is both a Python wrapper for VISA shared libraries but can also serve as a front-end for other VISA implementation such as *PyVISA-Py*.

1.1 User guide

This section of the documentation will focus on getting you started with PyVISA. The following sections will cover how to install and configure the library, how to communicate with your instrument and how to debug standard communications issues.

1.1.1 Installation

PyVISA is a frontend to the VISA library. It runs on Python 3.6+.

You can install it using pip:

```
$ pip install -U pyvisa
```

Backend

In order for PyVISA to work, you need to have a suitable backend. PyVISA includes a backend that wraps the National Instruments's VISA library. However, you need to download and install the library yourself (See *NI-VISA Installation*). There are multiple VISA implementations from different vendors. PyVISA is tested against National Instruments's VISA and Keysight IO Library Suite which can both be downloaded for free (you do not need a development environment only the driver library).

Warning: PyVISA works with 32- and 64- bit Python and can deal with 32- and 64-bit VISA libraries without any extra configuration. What PyVISA cannot do is open a 32-bit VISA library while running in 64-bit Python (or the other way around).

You need to make sure that the Python and VISA library have the same bitness

Alternatively, you can install PyVISA-Py which is a pure Python implementation of the VISA standard. You can install it using pip:

```
$ pip install -U pyvisa-py
```

Note: At the moment, *PyVISA-Py* implements only a limited subset of the VISA standard and does not support all protocols on all bus systems. Please refer to its documentation for more details.

Testing your installation

That's all! You can check that PyVISA is correctly installed by starting up python, and creating a ResourceManager:

```
>>> import pyvisa
>>> rm = pyvisa.ResourceManager()
>>> print(rm.list_resources())
```

If you encounter any problem, take a look at the *Miscellaneous questions*. There you will find the solutions to common problem as well as useful debugging techniques. If everything fails, feel free to open an issue in our issue tracker

Using the development version

You can install the latest development version (at your own risk) directly form GitHub:

```
$ pip install -U git+https://github.com/pyvisa/pyvisa.git
```

Note: If you have an old system installation of Python and you don't want to mess with it, you can try Anaconda. It is a free Python distribution by Continuum Analytics that includes many scientific packages.

1.1.2 Configuring the backend

Currently there are two backends available: The one included in pyvisa, which uses the IVI library (include NI-VISA, Keysight VISA, R&S VISA, tekVISA etc.), and the backend provided by pyvisa-py, which is a pure python implementation of the VISA library. If no backend is specified, pyvisa uses the IVI backend if any IVI library has been installed (see next section for details). Failing that, it uses the pyvisa-py backend.

You can also select a desired backend by passing a parameter to the ResourceManager, shown here for pyvisa-py:

```
>>> visa.ResourceManager('@py')
```

Alternatively it can also be selected by setting the environment variable PYVISA_LIBRARY. It takes the same values as the ResourceManager constructor.

Configuring the IVI backend

Note: The IVI backend requires that you install first the IVI-VISA library. For example you can use NI-VISA or any other library in your opinion. about NI-VISA get info here: (*NI-VISA Installation*)

In most cases PyVISA will be able to find the location of the shared visa library. If this does not work or you want to use another one, you need to provide the library path to the *ResourceManager* constructor:

```
>>> rm = ResourceManager('Path to library')
```

You can make this library the default for all PyVISA applications by using a configuration file called `.pyvisarc` (mind the leading dot) in your home directory.

Operating System	Location
Windows NT	`<root>\WINNT\Profiles\<username>`
Windows 2000, XP and 2003	`<root>\Documents and Settings\<username>`
Windows Vista, 7 or 8	`<root>\Users\<username>`
Mac OS X	`/Users/<username>`
Linux	`/home/<username>` (depends on the distro)

For example in Windows XP, place it in your user folder "Documents and Settings" folder, e.g. `C:\Documents and Settings\smith\.pyvisarc` if "smith" is the name of your login account.

This file has the format of an INI file. For example, if the library is at `/usr/lib/libvisa.so.7`, the file `.pyvisarc` must contain the following:

```
[Paths]

VISA library: /usr/lib/libvisa.so.7
```

Please note that *[Paths]* is treated case-sensitively.

To specify extra .dll search paths on Windows, use a *.pyvisarc* file like the following:

```
[Paths]

dll_extra_paths: C:\Program Files\Keysight\IO Libraries Suite\bin;C:\Program Files␣
↪(x86)\Keysight\IO Libraries Suite\bin
```

You can define a site-wide configuration file at `/usr/share/pyvisa/.pyvisarc` (It may also be `/usr/local/...` depending on the location of your Python). Under Windows, this file is usually placed at `c:\Python37\share\pyvisa\.pyvisarc`.

If you encounter any problem, take a look at the *Frequently asked questions*. There you will find the solutions to common problem as well as useful debugging techniques. If everything fails, feel free to open an issue in our issue tracker

1.1.3 Communicating with your instrument

Note: If you have been using PyVISA before version 1.5, you might want to read *Migrating from PyVISA < 1.5*.

An example

Let's go *in medias res* and have a look at a simple example:

```
>>> import pyvisa
>>> rm = pyvisa.ResourceManager()
>>> rm.list_resources()
('ASRL1::INSTR', 'ASRL2::INSTR', 'GPIB0::14::INSTR')
>>> my_instrument = rm.open_resource('GPIB0::14::INSTR')
>>> print(my_instrument.query('*IDN?'))
```

This example already shows the two main design goals of PyVISA: preferring simplicity over generality, and doing it the object-oriented way.

After importing `pyvisa`, we create a `ResourceManager` object. If called without arguments, PyVISA will prefer the default backend (IVI) which tries to find the VISA shared library for you. If it fails it will fall back to pyvisa-py if installed. You can check what backend is used and the location of the shared library used, if relevant, simply by:

```
>>> print(rm)
<ResourceManager('/path/to/visa.so')>
```

Note: In some cases, PyVISA is not able to find the library for you resulting in an `OSError`. To fix it, find the library path yourself and pass it to the ResourceManager constructor. You can also specify it in a configuration file as discussed in *Configuring the backend*.

Once that you have a `ResourceManager`, you can list the available resources using the `list_resources` method. The output is a tuple listing the *VISA resource names*. You can use a dedicated regular expression syntax to filter the instruments discovered by this method. The syntax is described in details in *`list_resources()`*. The default value is '?*::INSTR' which means that by default only instrument whose resource name ends with '::INSTR' are listed (in particular USB RAW resources and TCPIP SOCKET resources are not listed).

In this case, there is a GPIB instrument with instrument number 14, so you ask the `ResourceManager` to open "'GPIB0::14::INSTR'" and assign the returned object to the *my_instrument*.

Notice `open_resource` has given you an instance of `GPIBInstrument` class (a subclass of the more generic `Resource`).

```
>>> print(my_instrument)
<GPIBInstrument('GPIB::14')>
```

There many `Resource` subclasses representing the different types of resources, but you do not have to worry as the `ResourceManager` will provide you with the appropriate class. You can check the methods and attributes of each class in the *Resource classes*

Then, you query the device with the following message: `'\*IDN?'`. Which is the standard GPIB message for "what are you?" or – in some cases – "what's on your display at the moment?". `query` is a short form for a `write` operation to send a message, followed by a `read`.

So:

```
>>> my_instrument.query("*IDN?")
```

is the same as:

```
>>> my_instrument.write('*IDN?')
>>> print(my_instrument.read())
```

Note: You can access all the opened resources by calling `rm.list_opened_resources()`. This will return a list of `Resource`, however note that this list is not dynamically updated.

Getting the instrument configuration right

For most instruments, you actually need to properly configure the instrument so that it understands the message sent by the computer (in particular how to identifies the end of the commands) and so that computer knows when the instrument is done talking. If you don't you are likely to see a `VisaIOError` reporting a timeout.

For message based instruments (which covers most of the use cases), this usually consists in properly setting the *`read_termination`* and *`write_termination`* attribute of the resource. Resources have more attributes described in *Resources*, but for now we will focus on those two.

The first place to look for the values you should set for your instrument is the manual. The information you are looking is usually located close to the beginning of the IO operation section of the manual. If you cannot find the value, you can try to iterate through a couple of standard values but this is not recommended approach.

Once you have that information you can try to configure your instrument and start communicating as follows:

```
>>> my_instrument.read_termination = '\n'
>>> my_instrument.write_termination = '\n'
>>> my_instrument.query('*IDN?')
```

Here we use *'n'* known as 'line feed'. This is a common value, another one is *'r'* i.e. 'carriage return', and in some cases the null byte '0' is used.

In an ideal world, this will work and you will be able to get an answer from your instrument. If it does not, it means the settings are likely wrong (the documentation does not always shine by its clarity). In the following we will discuss common debugging tricks, if nothing works feel free to post on the PyVISA issue tracker. If you do be sure to describe in detail your setup and what you already attempted.

Note: The particular case of reading back large chunk of data either in ASCII or binary format is not discussed below but in *Reading and Writing values*.

Making sure the instrument understand the command

When using query, we are testing both writing to and reading from the instrument. The first thing to do is to try to identify if the issue occurs during the write or the read operation.

If your instrument has a front panel, you can check for errors (some instrument will display a transient message right after the read). If an error occurs, it may mean your command string contains a mistake or the instrument is using a different set of command (some instrument supports both a legacy set of commands and SCPI commands). If you see no error it means that either the instrument did not detect the end of your message or you just cannot read it. The next step is to determine in what situation we are.

To do so, you can look for a command that would produce a visible/measurable change on the instrument and send it. In the absence of errors, if the expected change did not occur it means the instrument did not understand that the command was complete. This points out to an issue with the *write_termination*. At this stage, you can go back to the manual (some instruments allow to switch between the recognized values), or try standards values (such as *'n'*, *'r'*, combination of those two, *'0'*).

Assuming you were able to confirm that the instrument understood the command you sent, it means the reading part is the issue, which is easier to troubleshoot. You can try different standard values for the *read_termination*, but if nothing works you can use the *read_bytes()* method. This method will read at most the number of bytes specified. So you can try reading one byte at a time till you encounter a time out. When that happens most likely the last character you read is the termination character. Here is a quick example:

```
my_instrument.write('*IDN?')
while True:
    print(my_instrument.read_bytes(1))
```

If *read_bytes()* times out on the first read, it actually means that the instrument did not answer. If the instrument is old it may be because your are too fast for it, so you can try waiting a bit before reading (using *time.sleep* from Python standard library). Otherwise, you either use a command that does not cause any answer or actually your write does not work (go back up a couple of paragraph).

Note: Some instruments may be slow in answering and may require you to either increase the timeout or specify a delay between the write and read operation. This can be done globally using *query_delay* or passing `delay=0.1` for example to wait 100 ms after writing before reading.

Note: When transferring large amount of data the total transfer time may exceed the timeout value in which case increasing the timeout value should fix the issue.

Note: It is possible to disable the use of teh termination character to detect the end of an input message by setting *read_termination* to `""`. Care has to be taken for the case of serial instrument for which the method used to determine the end of input is controlled by the `end_input` attribute and is set by default to use the termination character. To fully disable the use of the termination character its value should be changed.

The above focused on using only PyVISA, if you are running Windows, or MacOS you are likely to have access to third party tools that can help. Some tips to use them are given in the next section.

Note: Some instruments do not react well to a communication error, and you may have to restart it to get it to work again.

Using third-party softwares

The implementation of VISA from National Instruments and Keysight both come with tools (NIMax, Keysight Connection Expert) that can be used to figure out what is wrong with your communication setup.

In both cases, you can open an interactive communication session to your instrument and tune the settings using a GUI (which can make things easier). The basic procedure is the one described above, if you can make it work in one of those tools you should be able, in most cases, to get it to work in PyVISA. However if it does not work using those tools, it won't work in PyVISA.

For serial instruments (true or emulated over USB), you can also try to directly communicate with it using Putty or Tera Term on Windows, CoolTerm or Terminal / screen on macOS.

Hopefully those simple tips will allow you to get through. In some cases, it may not be the case and you are always welcome to ask for help (but realize that the maintainers are unlikely to have access to the instrument you are having trouble with).

1.1.4 A more complex example

The following example shows how to use SCPI commands with a Keithley 2000 multimeter in order to measure 10 voltages. After having read them, the program calculates the average voltage and prints it on the screen.

I'll explain the program step-by-step. First, we have to initialize the instrument:

```
>>> keithley = rm.open_resource("GPIB::12")
>>> keithley.write("*rst; status:preset; *cls")
```

Here, we create the instrument variable *keithley*, which is used for all further operations on the instrument. Immediately after it, we send the initialization and reset message to the instrument.

The next step is to write all the measurement parameters, in particular the interval time (500ms) and the number of readings (10) to the instrument. I won't explain it in detail. Have a look at an SCPI and/or Keithley 2000 manual.

```
>>> interval_in_ms = 500
>>> number_of_readings = 10
>>> keithley.write("status:measurement:enable 512; *sre 1")
>>> keithley.write("sample:count %d" % number_of_readings)
>>> keithley.write("trigger:source bus")
>>> keithley.write("trigger:delay %f" % (interval_in_ms / 1000.0))
>>> keithley.write("trace:points %d" % number_of_readings)
>>> keithley.write("trace:feed sense1; trace:feed:control next")
```

Okay, now the instrument is prepared to do the measurement. The next three lines make the instrument waiting for a trigger pulse, trigger it, and wait until it sends a "service request":

```
>>> keithley.write("initiate")
>>> keithley.assert_trigger()
>>> keithley.wait_for_srq()
```

With sending the service request, the instrument tells us that the measurement has been finished and that the results are ready for transmission. We could read them with *keithley.query("trace:data?")* however, then we'd get:

```
-000.0004E+0,-000.0005E+0,-000.0004E+0,-000.0007E+0,
-000.0000E+0,-000.0007E+0,-000.0008E+0,-000.0004E+0,
-000.0002E+0,-000.0005E+0
```

which we would have to convert to a Python list of numbers. Fortunately, the *query_ascii_values()* method does this work for us:

```
>>> voltages = keithley.query_ascii_values("trace:data?")
>>> print("Average voltage: ", sum(voltages) / len(voltages))
```

Finally, we should reset the instrument's data buffer and SRQ status register, so that it's ready for a new run. Again, this is explained in detail in the instrument's manual:

```
>>> keithley.query("status:measurement?")
>>> keithley.write("trace:clear; trace:feed:control next")
```

That's it. 18 lines of lucid code. (Well, SCPI is awkward, but that's another story.)

1.1.5 Reading and Writing values

Some instruments allow to transfer to and from the computer larger datasets with a single query. A typical example is an oscilloscope, which you can query for the whole voltage trace. Or an arbitrary wave generator to which you have to transfer the function you want to generate.

Basically, data like this can be transferred in two ways: in ASCII form (slow, but human readable) and binary (fast, but more difficult to debug).

PyVISA Message Based Resources have different methods for this called *read_ascii_values()*, *query_ascii_values()* and *read_binary_values()*, *query_binary_values()*.

Reading ASCII values

If your oscilloscope (open in the variable `inst`) has been configured to transfer data in **ASCII** when the `CURV?` command is issued, you can just query the values like this:

```
>>> values = inst.query_ascii_values('CURV?')
```

`values` will be `list` containing the values from the device.

In many cases you do not want a `list` but rather a different container type such as a `numpy.array`. You can of course cast the data afterwards like this:

```
>>> values = np.array(inst.query_ascii_values('CURV?'))
```

but sometimes it is much more efficient to avoid the intermediate list, and in this case you can just specify the container type in the query:

```
>>> values = inst.query_ascii_values('CURV?', container=numpy.array)
```

In `container`, you can have any callable/type that takes an iterable.

Note: When using numpy.array or numpy.ndarray, PyVISA will use numpy routines to optimize the conversion by avoiding the use of an intermediate representation.

Some devices transfer data in ASCII but not as decimal numbers but rather hex or oct. Or you might want to receive an array of strings. In that case you can specify a `converter`. For example, if you expect to receive integers as hex:

```
>>> values = inst.query_ascii_values('CURV?', converter='x')
```

`converter` can be one of the Python string formatting codes. But you can also specify a callable that takes a single argument if needed. The default converter is `'f'`.

Finally, some devices might return the values separated in an uncommon way. For example if the returned values are separated by a `'$'` you can do the following call:

```
>>> values = inst.query_ascii_values('CURV?', separator='$')
```

You can provide a function to takes a string and returns an iterable. Default value for the separator is `','` (comma)

Reading binary values

If your oscilloscope (open in the variable `inst`) has been configured to transfer data in **BINARY** when the `CURV?` command is issued, you need to know which type datatype (e.g. uint8, int8, single, double, etc) is being used. PyVISA use the same naming convention as the struct module.

You also need to know the *endianness*. PyVISA assumes little-endian as default. If you have doubles *d* in big endian the call will be:

```
>>> values = inst.query_binary_values('CURV?', datatype='d', is_big_endian=True)
```

You can also specify the output container type, just as it was shown before.

By default, PyVISA will assume that the data block is formatted according to the IEEE convention. If your instrument uses HP data block you can pass `header_fmt='hp'` to `read_binary_values`. If your instrument does not use any header for the data simply `header_fmt='empty'`.

By default PyVISA assumes, that the instrument will add the termination character at the end of the data block and actually makes sure it reads it to avoid issues. This behavior fits well a number of devices. However some devices omit the termination character, in which cases the operation will timeout. In this situation, first makes sure you can actually read from the instrument by reading the answer using the `read_raw` function (you may need to call it multiple time), and check that the advertized length of the block match what you get from your instrument (plus the header). If it is so, then you can safely pass `expect_termination=False`, and PyVISA will not look for a termination character at the end of the message.

If you can read without any problem from your instrument, but cannot retrieve the full message when using this method (VI_ERROR_CONN_LOST, VI_ERROR_INV_SETUP, or Python simply crashes), try passing different values for `chunk_size`` (the default is 20*1024). The underlying mechanism for this issue is not clear but changing ``chunk_size` has been used to work around it. Note that using larger chunk sizes for large transfer may result in a speed up of the transfer.

In some cases, the instrument may use a protocol that does not indicate how many bytes will be transferred. The Keithley 2000 for example always return the full buffer whose size is reported by the 'trace:points?' command. Since a binary block may contain the termination character, PyVISA need to know how many bytes to expect. For those case, you can pass the expected number of points using the `data_points` keyword argument. The number of bytes will be inferred from the datatype of the block.

Finally if you are reading a file for example and simply want to extract a bytes object, you can use the `"s"` datatype and pass `bytes` as container.

Writing ASCII values

To upload a function shape to arbitrary wave generator, the command might be `WLISt:WAVeform:DATA <waveform name>,<function data>` where `<waveform name>` tells the device under which name to store the data.

```
>>> values = list(range(100))
>>> inst.write_ascii_values('WLISt:WAVeform:DATA somename,', values)
```

Again, you can specify the converter code.

```
>>> inst.write_ascii_values('WLISt:WAVeform:DATA somename,', values, converter='x')
```

`converter` can be one of the Python string formatting codes. But you can also specify a callable that takes a single argument if needed. The default converter is `'f'`.

The separator can also be specified just like in `query_ascii_values`.

```
>>> inst.write_ascii_values('WLISt:WAVeform:DATA somename,', values, converter='x',␣
↪separator='$')
```

You can provide a function to takes a iterable and returns an string. Default value for the separator is `','` (comma)

Writing binary values

To upload a function shape to arbitrary wave generator, the command might be `WLISt:WAVeform:DATA <waveform name>,<function data>` where `<waveform name>` tells the device under which name to store the data.

```
>>> values = list(range(100))
>>> inst.write_binary_values('WLISt:WAVeform:DATA somename,', values)
```

Again you can specify the `datatype` and `endianness`.

```
>>> inst.write_binary_values('WLISt:WAVeform:DATA somename,', values, datatype='d',␣
↪is_big_endian=False)
```

If your data are already in a `bytes` object you can use the `"s"` format.

When things are not what they should be

PyVISA provides an easy way to transfer data from and to the device. The methods described above work fine for 99% of the cases but there is always a particular device that do not follow any of the standard protocols and is so different that cannot be adapted with the arguments provided above.

In those cases, you need to get the data:

```
>>> inst.write('CURV?')
>>> data = inst.read_raw()
```

and then you need to implement the logic to parse it.

Alternatively if the *read_raw* call fails you can try to read just a few bytes using:

```
>>> inst.write('CURV?')
>>> data = inst.read_bytes(1)
```

If this call fails it may mean that your instrument did not answer, either because it needs more time or because your first instruction was not understood.

1.1.6 Event handling

VISA supports generating events on the instrument side usually when a register change and handling then on the controller using two different mechanisms: - storing the events in a queue - calling a dedicated handler function registered for that purpose when the event occurs

PyVISA supports using both mechanism and tries to provide a convenient interface to both. Below we give a couple of example of how to use each mechanism (using a fictional instrument).

Waiting on events using a queue

First let's have a look at how to wait for an event to occur which will be stored in a queue.

```
from pyvisa import ResourceManager, constants

rm = ResourceManager

with rm.open_resource("TCPIP::192.168.0.2::INSTR") as instr:

    # Type of event we want to be notified about
    event_type = constants.EventType.service_request
    # Mechanism by which we want to be notified
    event_mech = constants.EventMechanism.queue

    instr.enable_event(event_type, event_mech)

    # Instrument specific code to enable service request
    # (for example on operation complete OPC)
    instr.write("*SRE 1")
    instr.write("INIT")

    # Wait for the event to occur
    response = instr.wait_on_event(event_type, 1000)
    assert response.event.event_type == event_type
    assert response.timed_out = False
    instr.disable_event(event_type, event_mech)
```

Let's examine the code. First, to avoid repeating ourselves, we store the type of event we want to be notified about and the mechanism we want to use to be notified. And we enable event notifications.

```
# Type of event we want to be notified about
event_type = constants.EventType.service_request
# Mechanism by which we want to be notified
event_mech = constants.EventMechanism.queue

instr.enable_event(event_type, event_mech)
```

Next we need to setup our instrument to generate the kind of event at the right time and start the operation that will lead to the event. For the sake of that example we are going to consider a Service Request event. Usually service request can be enabled for a range of register state, the details depending on the instrument. One useful case is to generate a service request when an operation is complete which is we are pretending to do here.

Finally we wait for the event to occur and we specify a timeout of 1000ms to avoid waiting for ever. Once we received the event we disable event handling.

Registering handlers for event

Rather than waiting for an event, it can sometimes be convenient to take immediate action when an event occurs, in which having the VISA library call directly a function can be useful. Let see how.

Note: One can enable event handling using both mechanisms (constants.EventMechanism.all)

```
from time import sleep
from pyvisa import ResourceManager, constants

rm = ResourceManager

def handle_event(resource, event, user_handle):
    resource.called = True
    print(f"Handled event {event.event_type} on {resource}")

with rm.open_resource("TCPIP::192.168.0.2::INSTR") as instr:

    instr.called = False

    # Type of event we want to be notified about
    event_type = constants.EventType.service_request
    # Mechanism by which we want to be notified
    event_mech = constants.EventMechanism.queue

    wrapped = instr.wrap_handler(handle_event)

    user_handle = instr.install_handler(event_type, wrapped, 42)
    instr.enable_event(event_type, event_mech, None)

    # Instrument specific code to enable service request
    # (for example on operation complete OPC)
    instr.write("*SRE 1")
    instr.write("INIT")

    while not instr.called:
        sleep(10)

    instr.disable_event(event_type, event_mech)
    instr.uninstall_handler(event_type, wrapped, user_handle)
```

Our handler function needs to have a specific signature to be used by VISA. The expected signature is (session, event_type, event_context, user_handle). This signature is not exactly convenient since it forces us to deal with a number of low-level details such session (ID of a resource in VISA) and event_context that serves the same purpose for events. One way to get a nicer interface is to wrap the handler using the *wrap_handler* method of the *Resource* object. The wrapped function is expected to have the following signature: (resource, event, user_handle) which the signature of our handler:

```
def handle_event(resource, event, user_handle):
    resource.called = True
    print(f"Handled event {event.event_type} on {resource}")
```

And before installing the handler, we wrap it:

```
wrapped = instr.wrap_handler(handle_event)
```

When wrapping a handler, you need to use the resource on which it is going to be installed to wrap it. Furthermore note that in order to uninstall a handler you need to keep the wrapped version around.

Next we install the handler and enable the event processing:

```
user_handle = instr.install_handler(event_type, wrapped, 42)
instr.enable_event(event_type, event_mech, None)
```

When installing a handler one can optionally, specify a user handle that will be passed to the handler. This handle can be used to identify which handler is called when registering the same handler multiple times on the same resource. That value may have to be converted by the backend. As a consequence the value passed to the handler may not the same as the value registered and its value will be to the backend dependent. For this reason you need to keep the converted value returned by install handler to uninstall the handler at a later time.

Note: In the case of ctwrapper that ships with PyVISA, the value is converted to an equivalent ctypes object (c_float for a float, c_int for an integer, etc)

1.1.7 Resources

A resource represents an instrument, e.g. a measurement device. There are multiple classes derived from resources representing the different available types of resources (eg. GPIB, Serial). Each contains the particular set of attributes an methods that are available by the underlying device.

You do not create this objects directly but they are returned by the *open_resource()* method of a *ResourceManager*. In general terms, there are two main groups derived from *Resource*, *MessageBasedResource* and *RegisterBasedResource*.

Note: The resource Python class to use is selected automatically from the resource name. However, you can force a Resource Python class:

```
>>> from pyvisa.resources import MessageBasedResource
>>> inst = rm.open('ASRL1::INSTR', resource_pyclass=MessageBasedResource)
```

The following sections explore the most common attributes of `Resource` and `MessageBased` (Serial, GPIB, etc) which are the ones you will encounter more often. For more information, refer to the *API*.

Attributes of Resource

session

Each communication channel to an instrument has a session handle which is unique. You can get this value:

```
>>> my_device.session
10442240
```

If the resource is closed, an exception will be raised:

```
>>> inst.close()
>>> inst.session
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
...
pyvisa.errors.InvalidSession: Invalid session handle. The resource might be closed.
```

timeout

Very most VISA I/O operations may be performed with a timeout. If a timeout is set, every operation that takes longer than the timeout is aborted and an exception is raised. Timeouts are given per instrument in **milliseconds**.

For all PyVISA objects, a timeout is set with

```
my_device.timeout = 25000
```

Here, `my_device` may be a device, an interface or whatever, and its timeout is set to 25 seconds. To set an **infinite** timeout, set it to `None` or `float('+inf')` or:

```
del my_device.timeout
```

To set it to **immediate**, set it to *0* or a negative value. (Actually, any value smaller than 1 is considered immediate)

Now every operation of the resource takes as long as it takes, even indefinitely if necessary.

Attributes of MessageBase resources

Chunk length

If you read data from a device, you must store it somewhere. Unfortunately, PyVISA must make space for the data *before* it starts reading, which means that it must know how much data the device will send. However, it doesn't know a priori.

Therefore, PyVISA reads from the device in *chunks*. Each chunk is 20 kilobytes long by default. If there's still data to be read, PyVISA repeats the procedure and eventually concatenates the results and returns it to you. Those 20 kilobytes are large enough so that mostly one read cycle is sufficient.

The whole thing happens automatically, as you can see. Normally you needn't worry about it. However, some devices don't like to send data in chunks. So if you have trouble with a certain device and expect data lengths larger than the default chunk length, you should increase its value by saying e.g.

```
my_instrument.chunk_size = 102400
```

This example sets it to 100 kilobytes.

Termination characters

Somehow the computer must detect when the device is finished with sending a message. It does so by using different methods, depending on the bus system. In most cases you don't need to worry about termination characters because the defaults are very good. However, if you have trouble, you may influence termination characters with PyVISA.

Termination characters may be one character or a sequence of characters. Whenever this character or sequence occurs in the input stream, the read operation is terminated and the read message is given to the calling application. The next read operation continues with the input stream immediately after the last termination sequence. In PyVISA, the termination characters are stripped off the message before it is given to you.

You may set termination characters for each instrument, e.g.

```
my_instrument.read_termination = '\r'
```

('r' is carriage return, usually appearing in the manuals as CR)

Alternatively you can give it when creating your instrument object:

```
my_instrument = rm.open_resource("GPIB::10", read_termination='\r')
```

The default value depends on the bus system. Generally, the sequence is empty, in particular for GPIB. For RS232 it's `\r`.

You can specify the character to add to each outgoing message using the `write_termination` attribute.

Note: Under the hood PyVISA manipulates several VISA attributes in a coherent manner. You can also access those directly if you need to see the :ref:visa-attr section below.

---

query_delay and *send_end*

There are two further options related to message termination, namely `send_end` and `query_delay`.

`send_end` is a boolean. If it's `True` (the default), the EOI line is asserted after each write operation, signalling the end of the operation. EOI is GPIB-specific but similar action is taken for other interfaces.

The argument `query_delay` is the time in seconds to wait after each write operation when performing a query. So you could write:

```
my_instrument = rm.open_resource("GPIB::10", send_end=False, delay=1.2)
```

This will set the delay to 1.2 seconds, and the EOI line is omitted. By the way, omitting EOI is *not* recommended, so if you omit it nevertheless, you should know what you're doing.

VISA attributes

In addition to the above mentioned attributes, you can access most of the VISA attributes as defined in the visa standard on your resources through properties. Those properties will take for you of converting Python values to values VISA values and hence simplify their manipulations. Some of those attributes also have lighter aliases that makes them easier to access as illustrated below:

```
from pyvisa import constants, ResourceManager
rm = ResourceManager()
instr = rm.open_resource('TCPIP0::1.2.3.4::56789::SOCKET')
instr.io_protocol = constants.VI_PROT_4882_STRS
# is equivalent to
instr.VI_ATTR_IO_PROT = constants.VI_PROT_4882_STRS
```

Note: To know the full list of attribute available on a resource you can inspect *visa_attributes_classes* or if you are using `pyvisa-shell` simply use the `attr` command.

You can also manipulate the VISA attributes using *get_visa_attribute* and *set_visa_attribute*. However you will have use the proper values (as defined in *pyvisa.constants*) both to access the attribute and to specify the value.

```
from pyvisa import constants, ResourceManager
rm = ResourceManager()
instr = rm.open_resource('TCPIP0::1.2.3.4::56789::SOCKET')
instr.set_visa_attribute(constants.VI_ATTR_SUPPRESS_END_EN, constants.VI_TRUE)
```

1.1.8 PyVISA Shell

The shell, moved into PyVISA from the Lantz Project is a text based user interface to interact with instruments. You can invoke it from the command-line:

```
pyvisa-shell
```

that will show something the following prompt:

```
Welcome to the VISA shell. Type help or ? to list commands.

(visa)
```

At any time, you can type `?` or `help` to get a list of valid commands:

```
(visa) help

Documented commands (type help <topic>):
========================================
EOF  attr  close  exit  help  list  open  query  read  timeout write

(visa) help list
List all connected resources.
```

Tab completion is also supported.

The most basic task is listing all connected devices:

```
(visa) list
( 0) ASRL1::INSTR
( 1) ASRL2::INSTR
( 2) USB0::0x1AB1::0x0588::DS1K00005888::INSTR
```

Each device/port is assigned a number that you can use for subsequent commands. Let's open comport 1:

```
(visa) open 0
ASRL1::INSTR has been opened.
You can talk to the device using "write", "read" or "query.
The default end of message is added to each message
(open) query *IDN?
Some Instrument, Some Company.
```

You can print timeout that is set for query/read operation:

```
(open) timeout
Timeout: 2000ms
```

Then also to change the timeout for example to 1500ms (1.5 sec):

```
(open) timeout 1500
Done
```

We can also get a list of all visa attributes:

```
(open) attr
+----------------------------+------------+----------------------------+-------------
↪-------------------------+
|          VISA name         |  Constant  |        Python name         |             ␣
↪    val                   |
+----------------------------+------------+----------------------------+-------------
↪-------------------------+
| VI_ATTR_ASRL_ALLOW_TRANSMIT | 1073676734 |       allow_transmit       |             ␣
↪     1                    |
|    VI_ATTR_ASRL_AVAIL_NUM   | 1073676460 |      bytes_in_buffer       |             ␣
↪     0                    |
|      VI_ATTR_ASRL_BAUD      | 1073676321 |         baud_rate          |             ␣
↪    9600                  |
|    VI_ATTR_ASRL_BREAK_LEN   | 1073676733 |        break_length        |             ␣
↪    250                   |
|   VI_ATTR_ASRL_BREAK_STATE  | 1073676732 |        break_state         |             ␣
↪     0                    |
|    VI_ATTR_ASRL_CONNECTED   | 1073676731 |                            |          VI_
↪ERROR_NSUP_ATTR          |
|    VI_ATTR_ASRL_CTS_STATE   | 1073676462 |                            |             ␣
↪     0                    |
|    VI_ATTR_ASRL_DATA_BITS   | 1073676322 |         data_bits          |             ␣
↪     8                    |
|    VI_ATTR_ASRL_DCD_STATE   | 1073676463 |                            |             ␣
↪     0                    |
|  VI_ATTR_ASRL_DISCARD_NULL  | 1073676464 |        discard_null        |             ␣
↪     0                    |
|    VI_ATTR_ASRL_DSR_STATE   | 1073676465 |                            |             ␣
↪     0                    |
|    VI_ATTR_ASRL_DTR_STATE   | 1073676466 |                            |             ␣
↪     1                    |
|     VI_ATTR_ASRL_END_IN     | 1073676467 |         end_input          |             ␣
↪     2                    |
|     VI_ATTR_ASRL_END_OUT    | 1073676468 |                            |             ␣
↪     0                    |
|   VI_ATTR_ASRL_FLOW_CNTRL   | 1073676325 |                            |             ␣
↪     0                    |
|     VI_ATTR_ASRL_PARITY     | 1073676323 |           parity           |             ␣
↪     0                    |
|  VI_ATTR_ASRL_REPLACE_CHAR  | 1073676478 |        replace_char        |             ␣
↪     0                    |
|    VI_ATTR_ASRL_RI_STATE    | 1073676479 |                            |             ␣
↪     0                    |
|    VI_ATTR_ASRL_RTS_STATE   | 1073676480 |                            |             ␣
↪     1                    |
|    VI_ATTR_ASRL_STOP_BITS   | 1073676324 |         stop_bits          |             ␣
↪     10                   |
|    VI_ATTR_ASRL_WIRE_MODE   | 1073676735 |                            |             ␣
↪    128                   |
|    VI_ATTR_ASRL_XOFF_CHAR   | 1073676482 |         xoff_char          |             ␣
↪     19                   |
|    VI_ATTR_ASRL_XON_CHAR    | 1073676481 |          xon_char          |             ␣
↪     17                   |
|     VI_ATTR_DMA_ALLOW_EN    | 1073676318 |         allow_dma          |             ␣
↪     0                    |
|    VI_ATTR_FILE_APPEND_EN   | 1073676690 |                            |             ␣
↪     0                    |
```

(continues on next page)

(continued from previous page)

```
|    VI_ATTR_INTF_INST_NAME    | 3221160169 |                              | ASRL1   (/
↪dev/cu.Bluetooth-PDA-Sync) |
|       VI_ATTR_INTF_NUM       | 1073676662 |       interface_number       |             ␣
↪      1                     |
|       VI_ATTR_INTF_TYPE      | 1073676657 |                              |             ␣
↪      4                     |
|       VI_ATTR_IO_PROT        | 1073676316 |         io_protocol          |             ␣
↪      1                     |
|   VI_ATTR_MAX_QUEUE_LENGTH   | 1073676293 |                              |             ␣
↪     50                     |
|   VI_ATTR_RD_BUF_OPER_MODE   | 1073676330 |                              |             ␣
↪      3                     |
|     VI_ATTR_RD_BUF_SIZE      | 1073676331 |                              |             ␣
↪     4096                   |
|      VI_ATTR_RM_SESSION      | 1073676484 |                              |             ␣
↪  3160976                   |
|      VI_ATTR_RSRC_CLASS      | 3221159937 |        resource_class        |             ␣
↪    INSTR                  |
|  VI_ATTR_RSRC_IMPL_VERSION   | 1073676291 |   implementation_version     |             ␣
↪  5243392                   |
|   VI_ATTR_RSRC_LOCK_STATE    | 1073676292 |         lock_state           |             ␣
↪      0                     |
|     VI_ATTR_RSRC_MANF_ID     | 1073676661 |                              |             ␣
↪     4086                   |
|    VI_ATTR_RSRC_MANF_NAME    | 3221160308 | resource_manufacturer_name   |             ␣
↪National Instruments        |
|      VI_ATTR_RSRC_NAME       | 3221159938 |        resource_name         |             ␣
↪ASRL1::INSTR                |
|  VI_ATTR_RSRC_SPEC_VERSION   | 1073676656 |         spec_version         |             ␣
↪  5243136                   |
|     VI_ATTR_SEND_END_EN      | 1073676310 |           send_end           |             ␣
↪      1                     |
|   VI_ATTR_SUPPRESS_END_EN    | 1073676342 |                              |             ␣
↪      0                     |
|       VI_ATTR_TERMCHAR       | 1073676312 |                              |             ␣
↪     10                     |
|     VI_ATTR_TERMCHAR_EN      | 1073676344 |                              |             ␣
↪      0                     |
|      VI_ATTR_TMO_VALUE       | 1073676314 |                              |             ␣
↪     2000                   |
|       VI_ATTR_TRIG_ID        | 1073676663 |                              |             ␣
↪     -1                     |
|   VI_ATTR_WR_BUF_OPER_MODE   | 1073676333 |                              |             ␣
↪      2                     |
|     VI_ATTR_WR_BUF_SIZE      | 1073676334 |                              |             ␣
↪     4096                   |
+------------------------------+------------+------------------------------+-------------
↪-------------------------+
```

To simplify the handling of VI_ATTR_TERMCHAR and VI_ATTR_TERMCHAR_EN, the command 'termchar' can be used. If only one character provided, it sets both read and write termination character to the same character. If two characters are provided, it sets read and write termination characters independently.

To setup termchar to 'r' (CR or ascii code 10):

```
(open) termchar CR
Done
```

To read what termchar is defined:

```
(open) termchar
Termchar read: CR write: CR
```

To setup read termchar to 'n' and write termchar to 'rn':

```
(open) termchar LF CRLF
Done
```

Supported termchar values are: CR ('r'), LF ('n'), CRLF ('rn') , NUL ('0'), None. None is used to disable termchar.

Finally, you can close the device:

```
(open) close
```

PyVisa Shell Backends

Based on available backend (see bellow for `info` command), it is possible to switch shell to use non-default backend via `-b BACKEND` or `--backend BACKEND`.

You can invoke:

```
pyvisa-shell -b sim
```

to use python-sim as backend instead of ni backend. This can be used for example for testing of python-sim configuration.

You can invoke:

```
pyvisa-shell -b py
```

uses python-py as backend instead of ivi backend, for situation when ivi not installed.

PyVisa Info

You can invoke it from the command-line:

```
pyvisa-info
```

that will print information to diagnose PyVISA, info about Machine, Python, backends, etc

```
Machine Details:
   Platform ID:    Windows
   Processor:      Intel64 Family 6
   ...
PyVISA Version: ...

Backends:
   ni:
      Version: 1.8 (bundled with PyVISA)
      ...
   py:
      Version: 0.2
      ...
  sim:
```

(continues on next page)

(continued from previous page)

```
    Version: 0.3
    Spec version: 1.1
```

Summary

Cool, right? It will be great to have a GUI similar to NI-MAX, but we leave that to be developed outside PyVISA. Want to help? Let us know!

1.1.9 VISA resource names

If you use the method *open_resource()*, you must tell this function the *VISA resource name* of the instrument you want to connect to. Generally, it starts with the bus type, followed by a double colon `"::"`, followed by the number within the bus. For example,

```
GPIB::10
```

denotes the GPIB instrument with the number 10. If you have two GPIB boards and the instrument is connected to board number 1, you must write

```
GPIB1::10
```

As for the bus, things like `"GPIB"`, `"USB"`, `"ASRL"` (for serial/parallel interface) are possible. So for connecting to an instrument at COM2, the resource name is

```
ASRL2
```

(Since only one instrument can be connected with one serial interface, there is no double colon parameter.) However, most VISA systems allow aliases such as `"COM2"` or `"LPT1"`. You may also add your own aliases.

The resource name is case-insensitive. It doesn't matter whether you say `"ASRL2"` or `"asrl2"`. For further information, I have to refer you to a comprehensive VISA description like http://www.ni.com/pdf/manuals/370423a.pdf.

VISA Resource Syntax and Examples

(This is adapted from the VISA manual)

The following table shows the grammar for the address string. Optional string segments are shown in square brackets ([ ]).

Interface	Syntax
ENET-Serial INSTR	ASRL[0]::host address::serial port::INSTR
GPIB INSTR	GPIB[board]::primary address[::secondary address][::INSTR]
GPIB INTFC	GPIB[board]::INTFC
PXI BACKPLANE	PXI[interface]::chassis number::BACKPLANE
PXI INSTR	PXI[bus]::device[::function][::INSTR]
PXI INSTR	PXI[interface]::bus-device[.function][::INSTR]
PXI INSTR	PXI[interface]::CHASSISchassis number::SLOTslot number[::FUNCfunction][::INSTR]
PXI MEMACC	PXI[interface]::MEMACC
Remote NI-VISA	visa://host address[:server port]/remote resource
Serial INSTR	ASRLboard[::INSTR]
TCPIP INSTR	TCPIP[board]::host address[::LAN device name][::INSTR]
TCPIP SOCKET	TCPIP[board]::host address::port::SOCKET
USB INSTR	USB[board]::manufacturer ID::model code::serial number[::USB interface number][::INSTR]
USB RAW	USB[board]::manufacturer ID::model code::serial number[::USB interface number]::RAW
VXI BACKPLANE	VXI[board][::VXI logical address]::BACKPLANE
VXI INSTR	VXI[board]::VXI logical address[::INSTR]
VXI MEMACC	VXI[board]::MEMACC
VXI SERVANT	VXI[board]::SERVANT

Use the GPIB keyword to establish communication with GPIB resources. Use the VXI keyword for VXI resources via embedded, MXIbus, or 1394 controllers. Use the ASRL keyword to establish communication with an asynchronous serial (such as RS-232 or RS-485) device. Use the PXI keyword for PXI and PCI resources. Use the TCPIP keyword for Ethernet communication.

The following table shows the default value for optional string segments.

Optional String Segments	Default Value
board	0
GPIB secondary address	none
LAN device name	inst0
PXI bus	0
PXI function	0
USB interface number	lowest numbered relevant interface

The following table shows examples of address strings:

Address String	Description
ASRL::1.2.3.4::2::INSTR	A serial device attached to port 2 of the ENET Serial controller at address 1.2.3.4.
ASRL1::INSTR	A serial device attached to interface ASRL1.
GPIB::1::0::INSTR	A GPIB device at primary address 1 and secondary address 0 in GPIB interface 0.
GPIB2::INTFC	Interface or raw board resource for GPIB interface 2.
PXI::15::INSTR	PXI device number 15 on bus 0 with implied function 0.
PXI::2::BACKPLANE	Backplane resource for chassis 2 on the default PXI system, which is interface 0.
PXI::CHASSIS1::SLOT3	PXI device in slot number 3 of the PXI chassis configured as chassis 1.
PXI0::2-12.1::INSTR	PXI bus number 2, device 12 with function 1.
PXI0::MEMACC	PXI MEMACC session.
TCPIP::dev.company.com::INSTR	A TCPIP device using VXI-11 or LXI located at the specified address. This uses the default LAN Device Name of inst0.
TCPIP0::1.2.3.4::999::SOCKET	Raw TCPIP access to port 999 at the specified IP address.
USB::0x1234::125::A22-5::INSTR	A USB Test & Measurement class device with manufacturer ID 0x1234, model code 125, and serial number A22-5. This uses the device's first available USBTMC interface. This is usually number 0.
USB::0x5678::0x33::SN999::1::RAW	A USB raw class device with manufacturer ID 0x5678, model code 0x33, and serial number SN999. This uses the device's interface number 1.
visa://hostname/ASRL1::INSTR	The resource ASRL1::INSTR on the specified remote system.
VXI::1::BACKPLANE	Mainframe resource for chassis 1 on the default VXI system, which is interface 0.
VXI::MEMACC	Board-level register access to the VXI interface.
VXI0::1::INSTR	A VXI device at logical address 1 in VXI interface VXI0.
VXI0::SERVANT	Servant/device-side resource for VXI interface 0.

1.2 Advanced topics

This section of the documentation will cover the internal details of PyVISA. In particular, it will explain in details how PyVISA manage backends.

1.2.1 Architecture

PyVISA implements convenient and Pythonic programming in three layers:

1. Low-level: A wrapper around the shared visa library.

   The wrapper defines the argument types and response types of each function, as well as the conversions between Python objects and foreign types.

   You will normally not need to access these functions directly. If you do, it probably means that we need to improve layer 2.

   All level 1 functions are **static methods** of *VisaLibraryBase*.

   > **Warning:** Notice however that low-level functions might not be present in all backends. For broader compatibility, do no use this layer. All the functionality should is available via the next layer.
   >
   > Alternative backends have no obligation to provide those functions.

2. Middle-level: A wrapping Python function for each function of the shared visa library.

These functions call the low-level functions, adding some code to deal with type conversions for functions that return values by reference. These functions also have comprehensive and Python friendly documentation.

You only need to access this layer if you want to control certain specific aspects of the VISA library which are not implemented by the corresponding resource class.

All level 2 functions are **bound methods** of *VisaLibraryBase*.

3. High-level: An object-oriented layer for *ResourceManager* and *Resource*.

   The *ResourceManager* implements methods to inspect connected resources. You also use this object to open other resources instantiating the appropriate *Resource* derived classes.

   Resource and the derived classes implement functions and attributes access to the underlying resources in a Pythonic way.

Most of the time you will only need to instantiate a *ResourceManager*. For a given resource, you will use the *open_resource()* method to obtain the appropriate object. If needed, you will be able to access the `VisaLibrary` object directly using the `visalib` attribute.

The `VisaLibrary` does the low-level calls. In the default IVI Backend, levels 1 and 2 are implemented in the same package called `pyvisa.ctwrapper` (which stands for ctypes wrapper). This package is included in PyVISA.

Other backends can be used just by passing the name of the backend to `ResourceManager` after the `@` symbol. See more information in *A frontend for multiple backends*.

Calling middle- and low-level functions

After you have instantiated the *ResourceManager*:

```
>>> import pyvisa
>>> rm = pyvisa.ResourceManager()
```

you can access the corresponding `VisaLibrary` instance under the `visalib` attribute.

As an example, consider the VISA function `viMapAddress`. It appears in the low-level layer as the static method `viMapAddress` of `visalib` attributed and also appears in the middle-level layer as `map_address`.

You can recognize low and middle-level functions by their names. Low-level functions carry the same name as in the shared library, and they are prefixed by **vi**. Middle-level functions have a friendlier, more pythonic but still recognizable name. Typically, camelCase names where stripped from the leading **vi** and changed to underscore separated lower case names. The docs about these methods is located here *API*.

Low-level

You can access the low-level functions directly exposed as static methods, for example:

```
>>> rm.visalib.viMapAddress(<here goes the arguments>)
```

To call this functions you need to know the function declaration and how to interface it to python. To help you out, the `VisaLibrary` object also contains middle-level functions.

It is very likely that you will need to access the VISA constants using these methods. You can find the information about these constants here *Constants module*

Middle-level

The `VisaLibrary` object exposes the middle-level functions which are one-to-one mapped from the foreign library as bound methods.

Each middle-level function wraps one low-level function. In this case:

```
>>> rm.visalib.map_address(<here goes the arguments>)
```

The calling convention and types are handled by the wrapper.

1.2.2 A frontend for multiple backends

A small historical note might help to make this section clearer. So bear with with me for a couple of lines. Originally PyVISA was a Python wrapper to the VISA library. More specifically, it was `ctypes` wrapper around the NI-VISA. This approach worked fine but made it difficult to develop other ways to communicate with instruments in platforms where NI-VISA was not available. Users had to change their programs to use other packages with different API.

Since 1.6, PyVISA is a frontend to VISA. It provides a nice, Pythonic API and can connect to multiple backends. Each backend exposes a class derived from VisaLibraryBase that implements the low-level communication. The ctypes wrapper around IVI-VISA is the default backend (called **ivi**) and is bundled with PyVISA for simplicity. In general, IVI-VISA can be NI-VISA, Keysight VISA, R&S VISA, tekVISA etc. By default, it calls the library that is installed on your system as VISA library.

You can specify the backend to use when you instantiate the resource manager using the `@` symbol. Remembering that **ivi** is the default, this:

```
>>> import pyvisa
>>> rm = pyvisa.ResourceManager()
```

is the same as this:

```
>>> import pyvisa
>>> rm = pyvisa.ResourceManager('@ivi')
```

You can still provide the path to the library if needed:

```
>>> import pyvisa
>>> rm = pyvisa.ResourceManager('/path/to/lib@ivi')
```

Under the hood, the *ResourceManager* looks for the requested backend and instantiate the VISA library that it provides.

PyVISA locates backends by name. If you do:

```
>>> import pyvisa
>>> rm = pyvisa.ResourceManager('@somename')
```

PyVISA will try to import a package/module named `pyvisa_somename` which should be installed in your system. This is a loosely coupled configuration free method. PyVISA does not need to know about any backend out there until you actually try to use it.

You can list the installed backends by running the following code in the command line:

```
pyvisa-info
```

Developing a new Backend

What does a minimum backend looks like? Quite simple:

```
from pyvisa.highlevel import VisaLibraryBase

class MyLibrary(VisaLibraryBase):
    pass

WRAPPER_CLASS = MyLibrary
```

Additionally you can provide a staticmethod named *get_debug_info* that should return a dictionary of debug information which is printed when you call `pyvisa-info`

An important aspect of developing a backend is knowing which VisaLibraryBase method to implement and what API to expose.

A **complete** implementation of a VISA Library requires a lot of functions (basically almost all level 2 functions as described in *Architecture* (there is also a complete list at the bottom of this page). But a working implementation does not require all of them.

As a **very minimum** set you need:

- **open_default_resource_manager**: returns a session to the Default Resource Manager resource.
- **open**: Opens a session to the specified resource.
- **close**: Closes the specified session, event, or find list.
- **list_resources**: Returns a tuple of all connected devices matching query.

(you can get the signature below or here *Visa Library*)

But of course you cannot do anything interesting with just this. In general you will also need:

- **get_attribute**: Retrieves the state of an attribute.
- **set_atribute**: Sets the state of an attribute.

If you need to start sending bytes to MessageBased instruments you will require:

- **read**: Reads data from device or interface synchronously.
- **write**: Writes data to device or interface synchronously.

For other usages or devices, you might need to implement other functions. Is really up to you and your needs.

These functions should raise a `pyvisa.errors.VisaIOError` or emit a `pyvisa.errors.VisaIOWarning` if necessary, and store error code on a per session basis. This can be done easily by calling *`handle_return_value()`* with the session and return value.

Complete list of level 2 functions to implement:

```
def read_memory(self, session, space, offset, width, extended=False):
def write_memory(self, session, space, offset, data, width, extended=False):
def move_in(self, session, space, offset, length, width, extended=False):
def move_out(self, session, space, offset, length, data, width, extended=False):
def peek(self, session, address, width):
def poke(self, session, address, width, data):
def assert_interrupt_signal(self, session, mode, status_id):
def assert_trigger(self, session, protocol):
def assert_utility_signal(self, session, line):
def buffer_read(self, session, count):
```

(continues on next page)

(continued from previous page)

```
def buffer_write(self, session, data):
def clear(self, session):
def close(self, session):
def disable_event(self, session, event_type, mechanism):
def discard_events(self, session, event_type, mechanism):
def enable_event(self, session, event_type, mechanism, context=None):
def flush(self, session, mask):
def get_attribute(self, session, attribute):
def gpib_command(self, session, data):
def gpib_control_atn(self, session, mode):
def gpib_control_ren(self, session, mode):
def gpib_pass_control(self, session, primary_address, secondary_address):
def gpib_send_ifc(self, session):
def in_8(self, session, space, offset, extended=False):
def in_16(self, session, space, offset, extended=False):
def in_32(self, session, space, offset, extended=False):
def in_64(self, session, space, offset, extended=False):
def install_handler(self, session, event_type, handler, user_handle):
def list_resources(self, session, query='?*::INSTR'):
def lock(self, session, lock_type, timeout, requested_key=None):
def map_address(self, session, map_space, map_base, map_size,
def map_trigger(self, session, trigger_source, trigger_destination, mode):
def memory_allocation(self, session, size, extended=False):
def memory_free(self, session, offset, extended=False):
def move(self, session, source_space, source_offset, source_width, destination_space,
def move_asynchronously(self, session, source_space, source_offset, source_width,
def move_in_8(self, session, space, offset, length, extended=False):
def move_in_16(self, session, space, offset, length, extended=False):
def move_in_32(self, session, space, offset, length, extended=False):
def move_in_64(self, session, space, offset, length, extended=False):
def move_out_8(self, session, space, offset, length, data, extended=False):
def move_out_16(self, session, space, offset, length, data, extended=False):
def move_out_32(self, session, space, offset, length, data, extended=False):
def move_out_64(self, session, space, offset, length, data, extended=False):
def open(self, session, resource_name,
def open_default_resource_manager(self):
def out_8(self, session, space, offset, data, extended=False):
def out_16(self, session, space, offset, data, extended=False):
def out_32(self, session, space, offset, data, extended=False):
def out_64(self, session, space, offset, data, extended=False):
def parse_resource(self, session, resource_name):
def parse_resource_extended(self, session, resource_name):
def peek_8(self, session, address):
def peek_16(self, session, address):
def peek_32(self, session, address):
def peek_64(self, session, address):
def poke_8(self, session, address, data):
def poke_16(self, session, address, data):
def poke_32(self, session, address, data):
def poke_64(self, session, address, data):
def read(self, session, count):
def read_asynchronously(self, session, count):
def read_stb(self, session):
def read_to_file(self, session, filename, count):
def set_attribute(self, session, attribute, attribute_state):
def set_buffer(self, session, mask, size):
def status_description(self, session, status):
```

(continues on next page)

(continued from previous page)

```
def terminate(self, session, degree, job_id):
def uninstall_handler(self, session, event_type, handler, user_handle=None):
def unlock(self, session):
def unmap_address(self, session):
def unmap_trigger(self, session, trigger_source, trigger_destination):
def usb_control_in(self, session, request_type_bitmap_field, request_id, request_
↪value,
def usb_control_out(self, session, request_type_bitmap_field, request_id, request_
↪value,
def vxi_command_query(self, session, mode, command):
def wait_on_event(self, session, in_event_type, timeout):
def write(self, session, data):
def write_asynchronously(self, session, data):
def write_from_file(self, session, filename, count):
```

1.2.3 Continuous integration setup

Testing PyVISA in a thorough manner is challenging due to the need to access both a VISA library implementation and actual instruments to test against. In their absence, tests are mostly limited to utility functions and infrastructure. Those limited tests are found at the root of the testsuite package of pyvisa. They are run, along with linters and documentation building on each commit using Github Actions.

Thanks to Keysight tools provided to PyVISA developers, it is also possible to test most capabilities of message based resources. However due to the hardware requirements for the build bots, those tests cannot be set up on conventional hosted CIs platform such as Travis, Azure, Github actions, etc.

Self-hosted builder can be used to run the tests requiring those tools. PyVISA developer have chosen to use Azure Pipelines to run self-hosted runners. This choice was based on the ease of use of Azure and the expected low maintenance the builder should require since the CIs proper is handled through Azure. Github Actions has also been considered but due to security reason, self-hosted runners should not run on forks and Github Actions does not currently provide a way to forbid running self-hosted runners on forks.

An Azure self-hosted runner has been set in place and will remain active till December 2020. This runner can only test TCPIP based resources. A new runner will be set up in the first trimester of 2021 with hopefully capabilities extended to USB::INSTR and GPIB resources.

The setup of the current runner is not perfect and the runner may go offline at times. If this happen, before December 2020, please contact @MatthieuDartiailh on Github.

Note: The current runner runs on Windows and uses conda. Due to the working of the activation scripts on Windows calls to *activate* or *conda activate* must be preceded by *call*.

1.3 Frequently asked questions

This section covers frequently asked questions in relation with PyVISA. You will find first miscellaneous questions and next a set of questions that requires more in depth answers.

1.3.1 Miscellaneous questions

Is *PyVISA* endorsed by National Instruments?

No. *PyVISA* is developed independently of National Instrument as a wrapper for the VISA library.

Who makes *PyVISA*?

PyVISA was originally programmed by Torsten Bronger and Gregor Thalhammer. It is based on earlier experiences by Thalhammer.

It was maintained from March 2012 to August 2013 by Florian Bauer. It was maintained from August 2013 to December 2017 by Hernan E. Grecco <hernan.grecco@gmail.com>. It is currently maintained by Matthieu Dartiailh <m.dartiailh@gmail.com>

Take a look at AUTHORS for more information

Is PyVISA thread-safe?

Yes, PyVISA is thread safe starting from version 1.6.

I have an error in my program and I am having trouble to fix it

PyVISA provides useful logs of all operations. Add the following commands to your program and run it again:

```
import pyvisa
pyvisa.log_to_screen()
```

I found a bug, how can I report it?

Please report it on the Issue Tracker, including operating system, python version and library version. In addition you might add supporting information by pasting the output of this command:

```
pyvisa-info
```

Error: Image not found

This error occurs when you have provided an invalid path for the VISA library. Check that the path provided to the constructor or in the configuration file

Error: Could not found VISA library

This error occurs when you have not provided a path for the VISA library and PyVISA is not able to find it for you. You can solve it by providing the library path to the `VisaLibrary` or `ResourceManager` constructor:

```
>>> visalib = VisaLibrary('/path/to/library')
```

or:

```
>>> rm = ResourceManager('Path to library')
```

or creating a configuration file as described in *Configuring the backend*.

Error: *visa* module has no attribute *ResourceManager*

The https://github.com/visa-sdk/visa-python provides a visa package that can conflict with `visa` module provided by PyVISA, which is why the `visa` module is deprecated and it is preferred to import `pyvisa` instead of `visa`. Both modules provides the same interface and no other changes should be needed.

Error: No matching architecture

This error occurs when you the Python architecture does not match the VISA architecture.

Note: PyVISA tries to parse the error from the underlying foreign function library to provide a more useful error message. If it does not succeed, it shows the original one.

In Mac OS X the original error message looks like this:

```
OSError: dlopen(/Library/Frameworks/visa.framework/visa, 6): no suitable image found.␣
↪ Did find:
    /Library/Frameworks/visa.framework/visa: no matching architecture in universal␣
↪wrapper
    /Library/Frameworks/visa.framework/visa: no matching architecture in universal␣
↪wrapper
```

In Linux the original error message looks like this:

```
OSError: Could not open VISA library:
    Error while accessing /usr/local/vxipnp/linux/bin/libvisa.so.7:/usr/local/vxipnp/
↪linux/bin/libvisa.so.7: wrong ELF class: ELFCLASS32
```

First, determine the details of your installation with the help of the following debug command:

```
pyvisa-info
```

You will see the 'bitness' of the Python interpreter and at the end you will see the list of VISA libraries that PyVISA was able to find.

The solution is to:

1. Install and use a VISA library matching your Python 'bitness'

   Download and install it from **National Instruments's VISA**. Run the debug command again to see if the new library was found by PyVISA. If not, create a configuration file as described in *Configuring the backend*.

   If there is no VISA library with the correct bitness available, try solution 2.

or

2. Install and use a Python matching your VISA library 'bitness'

   In Windows and Linux: Download and install Python with the matching bitness. Run your script again using the new Python

   In Mac OS X, Python is usually delivered as universal binary (32 and 64 bits).

   You can run it in 32 bit by running:

   ```
   arch -i386 python myscript.py
   ```

   or in 64 bits by running:

```
arch -x86_64 python myscript.py
```

You can create an alias by adding the following line

alias python32="arch -i386 python"

into your .bashrc or .profile or ~/.bash_profile (or whatever file depending on which shell you are using.)

You can also create a virtual environment for this.

OSError: Could not open VISA library: function 'viOpen' not found

Starting with Python 3.8, the .dll load behavior has changed on Windows (see https://docs.python.org/3/whatsnew/3.8.html#bpo-36085-whatsnew). This causes some versions of Keysight VISA to fail to load because it cannot find its .dll dependencies. You can solve it by creating a configuration file and setting *dll_extra_paths* as described in *Configuring the backend*.

VisaIOError: VI_ERROR_SYSTEM_ERROR: Unknown system error:

If you have an issue creating a pyvisa.ResourceManager object, first enable screen logging (pyvisa.log_to_screen()) to ensure it is correctly finding the dll files. If it is correctly finding the dlls, you may see an error similar to: * viOpenDefaultRM('<ViObject object at 0x000002B6CA4658C8>',) -> -1073807360 This issue was resolved by reinstalling python. It seems that something within the ctypes may have been corrupted. [https://github.com/pyvisa/pyvisa/issues/538]

Where can I get more information about VISA?

- The original VISA docs:
  - VISA specification (scroll down to the end)
  - VISA library specification
  - VISA specification for textual languages
- The very good VISA manuals from National Instruments's VISA:
  - NI-VISA User Manual
  - NI-VISA Programmer Reference Manual
  - NI-VISA help file in HTML

1.3.2 NI-VISA Installation

In every OS, the NI-VISA library bitness (i.e. 32- or 64-bit) has to match the Python bitness. So first you need to install a NI-VISA that works with your OS and then choose the Python version matching the installed NI-VISA bitness.

PyVISA includes a debugging command to help you troubleshoot this (and other things):

```
pyvisa-info
```

According to National Instruments, NI VISA **17.5** is available for the following platforms.

Note: If NI-VISA is not available for your system, take a look at the *Frequently asked questions*.

Mac OS X

Download NI-VISA for Mac OS X

Supports:

- Mac OS X 10.7.x x86 and x86-64
- Mac OS X 10.8.x

64-bit VISA applications are supported for a limited set of instrumentation buses. The supported buses are ENET-Serial, USB, and TCPIP. Logging VISA operations in NI I/O Trace from 64-bit VISA applications is not supported.

Windows

Download NI-VISA for Windows

Suports:

- Windows Server 2003 R2 (32-bit version only)
- Windows Server 2008 R2 (64-bit version only)
- Windows 8 x64 Edition (64-bit version)
- Windows 8 (32-bit version)
- Windows 7 x64 Edition (64-bit version)
- Windows 7 (32-bit version)
- Windows Vista x64 Edition (64-bit version)
- Windows Vista (32-bit version)
- Windows XP Service Pack 3

Support for Windows Server 2003 R2 may require disabling physical address extensions (PAE).

Linux

Download NI-VISA for Linux

Supports:

- openSUSE 12.2
- openSUSE 12.1
- Red Hat Enterprise Linux Desktop + Workstation 6
- Red Hat Enterprise Linux Desktop + Workstation 5
- Scientific Linux 6.x
- Scientific Linux 5.x

More details details can be found in the README of the installer.

Note: NI-VISA runs on other linux distros but the installation is more cumbersome. On Arch linux and related distributions, the AUR package ni-visa (early development) is known to work for the USB and TCPIP interfaces. Please note that you should restart after the installation for things to work properly.

1.3.3 Migrating from PyVISA < 1.5

Note: if you want PyVISA 1.4 compatibility use PyVISA 1.5 that provides Python 3 support, better visa library detection heuristics, Windows, Linux and OS X support, and no singleton object. PyVISA 1.6+ introduces a few compatibility breaks.

Some of these decisions were inspired by the `visalib` package as a part of Lantz

Short summary

PyVISA 1.5 has full compatibility with previous versions of PyVISA using the legacy module (changing some of the underlying implementation). But you are encouraged to do a few things differently if you want to keep up with the latest developments and be compatible with PyVISA > 1.5.

Indeed PyVISA 1.6 breaks compatibility to bring across a few good things.

If you are doing:

```
>>> import pyvisa
>>> keithley = pyvisa.instrument("GPIB::12")
>>> print(keithley.ask("*IDN?"))
```

change it to:

```
>>> import pyvisa
>>> rm = pyvisa.ResourceManager()
>>> keithley = rm.open_resource("GPIB::12")
>>> print(keithley.query("*IDN?"))
```

If you are doing:

```
>>> print(pyvisa.get_instruments_list())
```

change it to:

```
>>> print(rm.list_resources())
```

If you are doing:

```
>>> import pyvisa.vpp43 as vpp43
>>> vpp43.visa_library.load_library("/path/to/my/libvisa.so.7")
```

change it to:

```
>>> import pyvisa
>>> rm = pyvisa.ResourceManager("/path/to/my/libvisa.so.7")
>>> lib = rm.visalib
```

If you are doing::

```
>>> vpp43.lock(session)
```

change it to:

```
>>> lib.lock(session)
```

or better:

```
>>> resource.lock()
```

If you are doing::

```
>>> inst.term_chars = '\r'
```

change it to:

```
>>> inst.read_termination = '\r'
>>> inst.write_termination = '\r'
```

If you are doing::

```
>>> print(lib.status)
```

change it to:

```
>>> print(lib.last_status)
```

or even better, do it per resource:

```
>>> print(rm.last_status) # for the resource manager
>>> print(inst.last_status) # for a specific instrument
```

If you are doing::

```
>>> inst.timeout = 1  # Seconds
```

change it to:

```
>>> inst.timeout = 1000  # Milliseconds
```

As you see, most of the code shown above is making a few things explict. It adds 1 line of code (instantiating the ResourceManager object) which is not a big deal but it makes things cleaner.

If you were using `printf`, `queryf`, `scanf`, `sprintf` or `sscanf` of `vpp43`, rewrite as pure Python code (see below).

If you were using `Instrument.delay`, change your code or use `Instrument.query_delay` (see below).

A few alias has been created to ease the transition:

- ask -> query
- ask_delay -> query_delay
- get_instrument -> open_resource

A more detailed description

Dropped support for string related functions

The VISA library includes functions to search and manipulate strings such as `printf`, `queryf`, `scanf`, `sprintf` and `sscanf`. This makes sense as VISA involves a lot of string handling operations. The original PyVISA imple-

mentation wrapped these functions. But these operations are easily expressed in pure python and therefore were rarely used.

PyVISA 1.5 keeps these functions for backwards compatibility but they are removed in 1.6.

We suggest that you replace such functions by a pure Python version.

Isolated low-level wrapping module

In the original PyVISA implementation, the low level implementation (`vpp43`) was mixed with higher level constructs. The VISA library was wrapped using ctypes.

In 1.5, we refactored it as `ctwrapper`. This allows us to test the foreign function calls by isolating them from higher level abstractions. More importantly, it also allows us to build new low level modules that can be used as drop in replacements for `ctwrapper` in high level modules.

In 1.6, we made the `ResourceManager` the object exposed to the user. The type of the `VisaLibrary` can selected depending of the `library_path` and obtained from a plugin package.

We have two of such packages planned:

- a Mock module that allows you to test a PyVISA program even if you do not have VISA installed.
- a CFFI based wrapper. CFFI is new python package that allows easier and more robust wrapping of foreign libraries. It might be part of Python in the future.

PyVISA 1.5 keeps `vpp43` in the legacy subpackage (reimplemented on top of `ctwrapper`) to help with the migration. This module is gone in 1.6.

All functions that were present in `vpp43` are now present in `ctwrapper` but they take an additional first parameter: the foreign library wrapper.

We suggest that you replace `vpp43` by accessing the `VisaLibrary` object under the attribute visalib of the resource manager which provides all foreign functions as bound methods (see below).

No singleton objects

The original PyVISA implementation relied on a singleton, global objects for the library wrapper (named `visa_library`, an instance of the old `pyvisa.vpp43.VisaLibrary`) and the resource manager (named `resource_manager`, and instance of the old `pyvisa.visa.ResourceManager`). These were instantiated on import and the user could rebind to a different library using the `load_library` method. Calling this method however did not affect `resource_manager` and might lead to an inconsistent state.

There were additionally a few global structures such a `status` which stored the last status returned by the library and the warning context to prevent unwanted warnings.

In 1.5, there is a new `VisaLibrary` class and a new `ResourceManager` class (they are both in `pyvisa.highlevel`). The new classes are not singletons, at least not in the strict sense. Multiple instances of `VisaLibrary` and `ResourceManager` are possible, but only if they refer to different foreign libraries. In code, this means:

```
>>> lib1 = pyvisa.VisaLibrary("/path/to/my/libvisa.so.7")
>>> lib2 = pyvisa.VisaLibrary("/path/to/my/libvisa.so.7")
>>> lib3 = pyvisa.VisaLibrary("/path/to/my/libvisa.so.8")
>>> lib1 is lib2
True
>>> lib1 is lib3
False
```

Most of the time, you will not need access to a `VisaLibrary` object but to a `ResourceManager`. You can do:

```
>>> lib = pyvisa.VisaLibrary("/path/to/my/libvisa.so.7")
>>> rm = lib.resource_manager
```

or equivalently:

```
>>> rm = pyvisa.ResourceManager("/path/to/my/libvisa.so.7")
```

Note: If the path for the library is not given, the path is obtained from the user settings file (if exists) or guessed from the OS.

In 1.6, the state returned by the library is stored per resource. Additionally, warnings can be silenced by resource as well. You can access with the `last_status` property.

All together, these changes makes PyVISA thread safe.

VisaLibrary methods as way to call Visa functions

In the original PyVISA implementation, the `VisaLibrary` class was just having a reference to the ctypes library and a few functions.

In 1.5, we introduced a new `VisaLibrary` class (`pyvisa.highlevel`) which has every single low level function defined in `ctwrapper` as bound methods. In code, this means that you can do:

```
>>> import pyvisa
>>> rm = pyvisa.ResourceManager("/path/to/my/libvisa.so.7")
>>> lib = rm.visalib
>>> print(lib.read_stb(session))
```

(But it is very likely that you do not have to do it as the resource should have the function you need)

It also has every single VISA foreign function in the underlying library as static method. In code, this means that you can do:

```
>>> status = ctypes.c_ushort()
>>> ret lib.viReadSTB(session, ctypes.byref(status))
>>> print(ret.value)
```

Ask vs. query

Historically, the method `ask` has been used in PyVISA to do a `write` followed by a `read`. But in many other programs this operation is called `query`. Thereby we have decided to switch the name, keeping an alias to help with the transition.

However, `ask_for_values` has not been aliased to `query_values` because the API is different. `ask_for_values` still uses the old formatting API which is limited and broken. We suggest that you migrate everything to `query_values`

Seconds to milliseconds

The timeout is now in milliseconds (not in seconds as it was before). The reason behind this change is to make it coherent with all other VISA implementations out there. The C-API, LabVIEW, .NET: all use milliseconds. Using the

same units not only makes it easy to migrate to PyVISA but also allows to profit from all other VISA docs out there without extra cognitive effort.

Removal of Instrument.delay and added Instrument.query_delay

In the original PyVISA implementation, `Instrument` takes a `delay` argument that adds a pause after each write operation (This also can be changed using the `delay` attribute).

In PyVISA 1.6, `delay` is removed. Delays after write operations must be added to the application code. Instead, a new attribute and argument `query_delay` is available. This allows you to pause between `write` and ``read` operations inside `query`. Additionally, `query` takes an optional argument called `query` allowing you to change it for each method call.

Deprecated term_chars and automatic removal of CR + LF

In the original PyVISA implementation, `Instrument` takes a `term_chars` argument to change at the read and write termination characters. If this argument is `None`, `CR + LF` is appended to each outgoing message and not expected for incoming messages (although removed if present).

In PyVISA 1.6, `term_chars` is replaced by `read_termination` and ``write_termination`. In this way, you can set independently the termination for each operation. Automatic removal of `CR + LF` is also gone in 1.6.

1.3.4 Contributing to PyVISA

You can contribute in different ways:

Report issues

You can report any issues with the package, the documentation to the PyVISA issue tracker. Also feel free to submit feature requests, comments or questions. In some cases, platform specific information is required. If you think this is the case, run the following command and paste the output into the issue:

```
pyvisa-info
```

It is useful that you also provide the log output. To obtain it, add the following lines to your code:

```
import pyvisa
pyvisa.log_to_screen()
```

If your issue concern a specific instrument please be sure to indicate the manufacturer and the model.

Contribute code

To contribute fixes, code or documentation to PyVISA, send us a patch, or fork PyVISA in github and submit the changes using a pull request.

You can also get the code from PyPI or GitHub. You can clone the public repository:

```
$ git clone git://github.com/pyvisa/pyvisa.git
```

Once you have a copy of the source, you can embed it in your Python package, or install it in develop mode easily:

```
$ python setup.py develop
```

Installing in development mode means that any change you make will be immediately reflected when you run pyvisa.

PyVISA uses a number of tools to ensure a consistent code style and quality. The code is checked as part of the CIs system but you may want to run those locally before submitting a patch. You have multiple options to do so:

- You can install *pre-commit* (using pip for example) and run:

```
$pre-commit install
```

This will run all the above mentioned tools run when you commit your changes.

- Install and run each tool independently. You can install all of them using pip and the *dev_requirements.txt* file. You can a look at the CIs configurations (in .github/workflows/ci.yml). Thoses tools are:
  - black: Code formatting
  - isort: Import sorting
  - flake8: Code quality
  - mypy: Typing

Finally if editing docstring, please note that PyVISA uses Numpy style docstring. In order to build the documentation you will need to install *sphinx* and *sphinx_rtd_theme*. Both are listed in *dev_requirements.txt*.

Note: If you have an old system installation of Python and you don't want to mess with it, you can try Anaconda. It is a free Python distribution by Continuum Analytics that includes many scientific packages.

Contributing to an existing backend

Backends are the central piece of PyVISA as they provide the low level communication over the different interfaces. There a couple of backends in the wild which can use your help. Look them up in PyPI (try *pyvisa* in the search box) and see where you can help.

Contributing a new backend

If you think there is a new way that low level communication can be achieved, go for it. You can use any of the existing backends as a template or start a thread in the issue tracker and we will be happy to help you.

1.4 API

1.4.1 Visa Library

class pyvisa.highlevel.**VisaLibraryBase**

Base for VISA library classes.

A class derived from *VisaLibraryBase* library provides the low-level communication to the underlying devices providing Pythonic wrappers to VISA functions. But not all derived class must/will implement all methods. Even if methods are expected to return the status code they are expected to raise the appropriate exception when an error ocurred since this is more Pythonic.

The default VisaLibrary class is `pyvisa.ctwrapper.highlevel.IVIVisaLibrary`, which implements a ctypes wrapper around the IVI-VISA library. Certainly, IVI-VISA can be NI-VISA, Keysight VISA, R&S VISA, tekVISA etc.

In general, you should not instantiate it directly. The object exposed to the user is the *pyvisa.highlevel.ResourceManager*. If needed, you can access the VISA library from it:

```
>>> import pyvisa
>>> rm = pyvisa.ResourceManager("/path/to/my/libvisa.so.7")
>>> lib = rm.visalib
```

assert_interrupt_signal(*session: NewType.<locals>.new_type*, *mode: pyvisa.constants.AssertSignalInterrupt*, *status_id: int*) → pyvisa.constants.StatusCode

Asserts the specified interrupt or signal.

Corresponds to viAssertIntrSignal function of the VISA library.

Parameters

- **session** (*VISASession*) – Unique logical identifier to a session.
- **mode** (*constants.AssertSignalInterrupt*) – How to assert the interrupt.
- **status_id** (*int*) – Status value to be presented during an interrupt acknowledge cycle.

Returns Return value of the library call.

Return type *StatusCode*

assert_trigger(*session: NewType.<locals>.new_type*, *protocol: pyvisa.constants.TriggerProtocol*) → pyvisa.constants.StatusCode

Assert software or hardware trigger.

Corresponds to viAssertTrigger function of the VISA library.

Parameters

- **session** (*VISASession*) – Unique logical identifier to a session.
- **protocol** (*constants.TriggerProtocol*) – Trigger protocol to use during assertion.

Returns Return value of the library call.

Return type *StatusCode*

assert_utility_signal(*session: NewType.<locals>.new_type*, *line: pyvisa.constants.UtilityBusSignal*) → pyvisa.constants.StatusCode

Assert or deassert the specified utility bus signal.

Corresponds to viAssertUtilSignal function of the VISA library.

Parameters

- **session** (*VISASession*) – Unique logical identifier to a session.
- **line** (*constants.UtilityBusSignal*) – Specifies the utility bus signal to assert.

Returns Return value of the library call.

Return type *StatusCode*

buffer_read(*session: NewType.<locals>.new_type*, *count: int*) → Tuple[bytes, pyvisa.constants.StatusCode]

Reads data through the use of a formatted I/O read buffer.

The data can be read from a device or an interface.

Corresponds to viBufRead function of the VISA library.

Parameters

- **session** (*VISASession Unique logical identifier to a session.*) –
- **count** (*int*) – Number of bytes to be read.

Returns

- *dbytes* – Data read
- *StatusCode* – Return value of the library call.

buffer_write(*session: NewType.<locals>.new_type*, *data: bytes*) → Tuple[int, pyvisa.constants.StatusCode]

Writes data to a formatted I/O write buffer synchronously.

Corresponds to viBufWrite function of the VISA library.

Parameters

- **session** (*VISASession*) – Unique logical identifier to a session.
- **data** (*bytes*) – Data to be written.

Returns

- *int* – number of written bytes
- *StatusCode* – return value of the library call.

clear(*session: NewType.<locals>.new_type*) → pyvisa.constants.StatusCode

Clears a device.

Corresponds to viClear function of the VISA library.

Parameters **session** (*VISASession*) – Unique logical identifier to a session.

Returns Return value of the library call.

Return type *StatusCode*

close(*session: Union[NewType.<locals>.new_type, NewType.<locals>.new_type, NewType.<locals>.new_type]*) → pyvisa.constants.StatusCode

Closes the specified session, event, or find list.

Corresponds to viClose function of the VISA library.

Parameters **session** (*Union[VISASession, VISAEventContext, VISARMSession]*) – Unique logical identifier to a session, event, resource manager.

Returns Return value of the library call.

Return type *StatusCode*

disable_event(*session: NewType.<locals>.new_type*, *event_type: pyvisa.constants.EventType*, *mechanism: pyvisa.constants.EventMechanism*) → pyvisa.constants.StatusCode

Disable notification for an event type(s) via the specified mechanism(s).

Corresponds to viDisableEvent function of the VISA library.

Parameters

- **session** (*VISASession*) – Unique logical identifier to a session.

- **event_type** (*constants.EventType*) – Event type.
- **mechanism** (*constants.EventMechanism*) – Event handling mechanisms to be disabled.

Returns Return value of the library call.

Return type *StatusCode*

discard_events(*session: NewType.<locals>.new_type*, *event_type: pyvisa.constants.EventType*, *mechanism: pyvisa.constants.EventMechanism*) → pyvisa.constants.StatusCode

Discard event occurrences for a given type and mechanisms in a session.

Corresponds to viDiscardEvents function of the VISA library.

Parameters

- **session** (*VISASession*) – Unique logical identifier to a session.
- **event_type** (*constans.EventType*) – Logical event identifier.
- **mechanism** (*constants.EventMechanism*) – Specifies event handling mechanisms to be discarded.

Returns Return value of the library call.

Return type *StatusCode*

enable_event(*session: NewType.<locals>.new_type*, *event_type: pyvisa.constants.EventType*, *mechanism: pyvisa.constants.EventMechanism*, *context: None = None*) → pyvisa.constants.StatusCode

Enable event occurrences for specified event types and mechanisms in a session.

Corresponds to viEnableEvent function of the VISA library.

Parameters

- **session** (*VISASession*) – Unique logical identifier to a session.
- **event_type** (*constants.EventType*) – Logical event identifier.
- **mechanism** (*constants.EventMechanism*) – Specifies event handling mechanisms to be enabled.
- **context** (*None, optional*) – Unused parameter...

Returns Return value of the library call.

Return type *StatusCode*

flush(*session: NewType.<locals>.new_type*, *mask: pyvisa.constants.BufferOperation*) → pyvisa.constants.StatusCode

Flush the specified buffers.

The buffers can be associated with formatted I/O operations and/or serial communication.

Corresponds to viFlush function of the VISA library.

Parameters

- **session** (*VISASession*) – Unique logical identifier to a session.
- **mask** (*constants.BufferOperation*) – Specifies the action to be taken with flushing the buffer. The values can be combined using the | operator. However multiple operations on a single buffer cannot be combined.

Returns Return value of the library call.

Return type *StatusCode*

get_attribute(*session: Union[NewType.<locals>.new_type, NewType.<locals>.new_type, NewType.<locals>.new_type], attribute: Union[pyvisa.constants.ResourceAttribute, pyvisa.constants.EventAttribute]*) → Tuple[Any, pyvisa.constants.StatusCode]

Retrieves the state of an attribute.

Corresponds to viGetAttribute function of the VISA library.

Parameters

- **session** (*Union[VISASession, VISAEventContext]*) – Unique logical identifier to a session, event, or find list.
- **attribute** (*Union[constants.ResourceAttribute, constants.EventAttribute]*) – Resource or event attribute for which the state query is made.

Returns

- *Any* – State of the queried attribute for a specified resource
- *StatusCode* – Return value of the library call.

get_buffer_from_id(*job_id: NewType.<locals>.new_type*) → Optional[SupportsBytes]

Retrieve the buffer associated with a job id created in read_asynchronously

Parameters **job_id** (*VISAJobID*) – Id of the job for which to retrieve the buffer.

Returns Buffer in which the data are stored or None if the job id is not associated with any job.

Return type Optional[SupportsBytes]

static get_debug_info() → Union[Iterable[str], Dict[str, Union[str, Dict[str, Any]]]]

Override to return an iterable of lines with the backend debug details.

get_last_status_in_session(*session: Union[NewType.<locals>.new_type, NewType.<locals>.new_type]*) → pyvisa.constants.StatusCode

Last status in session.

Helper function to be called by resources properties.

static get_library_paths() → Iterable[pyvisa.util.LibraryPath]

Override to list the possible library_paths if no path is specified.

gpib_command(*session: NewType.<locals>.new_type, data: bytes*) → Tuple[int, pyvisa.constants.StatusCode]

Write GPIB command bytes on the bus.

Corresponds to viGpibCommand function of the VISA library.

Parameters

- **session** (*VISASession*) – Unique logical identifier to a session.
- **data** (*bytes*) – Data to write.

Returns

- *int* – Number of written bytes
- *StatusCode* – Return value of the library call.

gpib_control_atn(*session: NewType.<locals>.new_type, mode: pyvisa.constants.ATNLineOperation*) → pyvisa.constants.StatusCode

Specifies the state of the ATN line and the local active controller state.

Corresponds to viGpibControlATN function of the VISA library.

Parameters

- **session** (*VISASession*) – Unique logical identifier to a session.
- **mode** (*constants.ATNLineOperation*) – State of the ATN line and optionally the local active controller state.

Returns Return value of the library call.

Return type *StatusCode*

gpib_control_ren(*session: NewType.<locals>.new_type*, *mode: pyvisa.constants.RENLineOperation*) → pyvisa.constants.StatusCode

Controls the state of the GPIB Remote Enable (REN) interface line.

Optionally the remote/local state of the device can also be set.

Corresponds to viGpibControlREN function of the VISA library.

Parameters

- **session** (*VISASession*) – Unique logical identifier to a session.
- **mode** (*constants.RENLineOperation*) – State of the REN line and optionally the device remote/local state.

Returns Return value of the library call.

Return type *StatusCode*

gpib_pass_control(*session: NewType.<locals>.new_type*, *primary_address: int*, *secondary_address: int*) → pyvisa.constants.StatusCode

Tell a GPIB device to become controller in charge (CIC).

Corresponds to viGpibPassControl function of the VISA library.

Parameters

- **session** (*VISASession*) – Unique logical identifier to a session.
- **primary_address** (*int*) – Primary address of the GPIB device to which you want to pass control.
- **secondary_address** (*int*) – Secondary address of the targeted GPIB device. If the targeted device does not have a secondary address, this parameter should contain the value Constants.VI_NO_SEC_ADDR.

Returns Return value of the library call.

Return type *StatusCode*

gpib_send_ifc(*session: NewType.<locals>.new_type*) → pyvisa.constants.StatusCode

Pulse the interface clear line (IFC) for at least 100 microseconds.

Corresponds to viGpibSendIFC function of the VISA library.

Parameters **session** (*VISASession*) – Unique logical identifier to a session.

Returns Return value of the library call.

Return type *StatusCode*

handle_return_value(*session: Union[NewType.<locals>.new_type, NewType.<locals>.new_type, NewType.<locals>.new_type, None]*, *status_code: int*) → pyvisa.constants.StatusCode

Helper function handling the return code of a low-level operation.

Used when implementing concrete subclasses of VISALibraryBase.

handlers = None
Contains all installed event handlers. Its elements are tuples with four elements: The handler itself (a Python callable), the user handle (in any format making sense to the lower level implementation, ie as a ctypes object for the ctypes backend) and the handler again, this time in a format meaningful to the backend (ie as a ctypes object created with CFUNCTYPE for the ctypes backend) and the event type.

ignore_warning(*session: Union[NewType.<locals>.new_type, NewType.<locals>.new_type], *warnings_constants*) → Iterator[T_co]
Ignore warnings for a session for the duration of the context.

Parameters

- **session** (*Union[VISASession, VISARMSession]*) – Unique logical identifier to a session.
- **warnings_constants** (StatusCode) – Constants identifying the warnings to ignore.

in_16(*session: NewType.<locals>.new_type*, *space: pyvisa.constants.AddressSpace*, *offset: int*, *extended: bool = False*) → Tuple[int, pyvisa.constants.StatusCode]
Reads in an 16-bit value from the specified memory space and offset.

Corresponds to viIn16* function of the VISA library.

Parameters

- **session** (*VISASession*) – Unique logical identifier to a session.
- **space** (*constants.AddressSpace*) – Specifies the address space.
- **offset** (*int*) – Offset (in bytes) of the address or register from which to read.
- **extended** (*bool, optional*) – Use 64 bits offset independent of the platform, False by default.

Returns

- *int* – Data read from memory
- *StatusCode* – Return value of the library call.

in_32(*session: NewType.<locals>.new_type*, *space: pyvisa.constants.AddressSpace*, *offset: int*, *extended: bool = False*) → Tuple[int, pyvisa.constants.StatusCode]
Reads in an 32-bit value from the specified memory space and offset.

Corresponds to viIn32* function of the VISA library.

Parameters

- **session** (*VISASession*) – Unique logical identifier to a session.
- **space** (*constants.AddressSpace*) – Specifies the address space.
- **offset** (*int*) – Offset (in bytes) of the address or register from which to read.
- **extended** (*bool, optional*) – Use 64 bits offset independent of the platform, False by default.

Returns

- *int* – Data read from memory
- *StatusCode* – Return value of the library call.

in_64(*session: NewType.<locals>.new_type*, *space: pyvisa.constants.AddressSpace*, *offset: int*, *extended: bool = False*) → Tuple[int, pyvisa.constants.StatusCode]
Reads in an 64-bit value from the specified memory space and offset.

Corresponds to viIn64* function of the VISA library.

Parameters

- **session** (*VISASession*) – Unique logical identifier to a session.
- **space** (*constants.AddressSpace*) – Specifies the address space.
- **offset** (*int*) – Offset (in bytes) of the address or register from which to read.
- **extended** (*bool, optional*) – Use 64 bits offset independent of the platform, False by default.

Returns

- *int* – Data read from memory
- *StatusCode* – Return value of the library call.

in_8(*session: NewType.<locals>.new_type*, *space: pyvisa.constants.AddressSpace*, *offset: int*, *extended: bool = False*) → Tuple[int, pyvisa.constants.StatusCode]

Reads in an 8-bit value from the specified memory space and offset.

Corresponds to viIn8* function of the VISA library.

Parameters

- **session** (*VISASession*) – Unique logical identifier to a session.
- **space** (*constants.AddressSpace*) – Specifies the address space.
- **offset** (*int*) – Offset (in bytes) of the address or register from which to read.
- **extended** (*bool, optional*) – Use 64 bits offset independent of the platform, False by default.

Returns

- *int* – Data read from memory
- *StatusCode* – Return value of the library call.

install_handler(*session: NewType.<locals>.new_type, event_type: pyvisa.constants.EventType, handler: Callable[[NewType.<locals>.new_type, pyvisa.constants.EventType, NewType.<locals>.new_type, Any], None], user_handle: Any*) → Tuple[Callable[[NewType.<locals>.new_type, pyvisa.constants.EventType, NewType.<locals>.new_type, Any], None], Any, Any, pyvisa.constants.StatusCode]

Install handlers for event callbacks.

Corresponds to viInstallHandler function of the VISA library.

Parameters

- **session** (*VISASession*) – Unique logical identifier to a session.
- **event_type** (*constants.EventType*) – Logical event identifier.
- **handler** (*VISAHandler*) – Reference to a handler to be installed by a client application.
- **user_handle** (*Any*) – Value specified by an application that can be used for identifying handlers uniquely for an event type.

Returns

- **handler** (*VISAHandler*) – Handler to be installed by a client application.

- *converted_user_handle* – Converted user handle to match the underlying library. This version of the handle should be used in further call to the library.
- *converted_handler* – Converted version of the handler satisfying to backend library.
- **status_code** (*StatusCode*) – Return value of the library call

install_visa_handler(*session: NewType.<locals>.new_type, event_type: pyvisa.constants.EventType, handler: Callable[[NewType.<locals>.new_type, pyvisa.constants.EventType, NewType.<locals>.new_type, Any], None], user_handle: Any = None*) → Any

Installs handlers for event callbacks.

Parameters

- **session** (*VISASession*) – Unique logical identifier to a session.
- **event_type** (*constants.EventType,*) – Logical event identifier.
- **handler** (*VISAHandler*) – Handler to be installed by a client application.
- **user_handle** – A value specified by an application that can be used for identifying handlers uniquely for an event type.

Returns Converted user handle to match the underlying library. This version of the handle should be used in further call to the library.

Return type converted_user_handle

issue_warning_on = None

Set error codes on which to issue a warning.

last_status

Last return value of the library.

library_path = None

Path to the VISA library used by this instance

list_resources(*session: NewType.<locals>.new_type*, *query: str = '?*::INSTR'*) → Tuple[str, ...]

Return a tuple of all connected devices matching query.

Parameters

- **session** (*VISARMSession*) – Unique logical identifier to the resource manager session.
- **query** (*str*) – Regular expression used to match devices.

Returns Resource names of all the connected devices matching the query.

Return type Tuple[str, ..]

lock(*session: NewType.<locals>.new_type*, *lock_type: pyvisa.constants.Lock*, *timeout: int*, *requested_key: Optional[str] = None*) → Tuple[str, pyvisa.constants.StatusCode]

Establishes an access mode to the specified resources.

Corresponds to viLock function of the VISA library.

Parameters

- **session** (*VISASession*) – Unique logical identifier to a session.
- **lock_type** (*constants.Lock*) – Specifies the type of lock requested.
- **timeout** (*int*) – Absolute time period (in milliseconds) that a resource waits to get unlocked by the locking session before returning an error.

- **requested_key** (*Optional[str], optional*) – Requested locking key in the case of a shared lock. For an exclusive lock it should be None.

Returns

- *Optional[str]* – Key that can then be passed to other sessions to share the lock, or None for an exclusive lock.
- *StatusCode* – Return value of the library call.

map_address(*session: NewType.<locals>.new_type*, *map_space: pyvisa.constants.AddressSpace*, *map_base: int*, *map_size: int*, *access: typing_extensions.Literal[False][False] = False*, *suggested: Optional[int] = None*) → Tuple[int, pyvisa.constants.StatusCode]

Maps the specified memory space into the process's address space.

Corresponds to viMapAddress function of the VISA library.

Parameters

- **session** (*VISASession*) – Unique logical identifier to a session.
- **map_space** (*constants.AddressSpace*) – Specifies the address space to map.
- **map_base** (*int*) – Offset (in bytes) of the memory to be mapped.
- **map_size** (*int*) – Amount of memory to map (in bytes).
- **access** (*False*) – Unused parameter.
- **suggested** (*Optional[int], optional*) – If not None, the operating system attempts to map the memory to the address specified. There is no guarantee, however, that the memory will be mapped to that address. This operation may map the memory into an address region different from the suggested one.

Returns

- *int* – Address in your process space where the memory was mapped
- *StatusCode* – Return value of the library call.

map_trigger(*session: NewType.<locals>.new_type*, *trigger_source: pyvisa.constants.InputTriggerLine*, *trigger_destination: pyvisa.constants.OutputTriggerLine*, *mode: None = None*) → pyvisa.constants.StatusCode

Map the specified trigger source line to the specified destination line.

Corresponds to viMapTrigger function of the VISA library.

Parameters

- **session** (*VISASession*) – Unique logical identifier to a session.
- **trigger_source** (*constants.InputTriggerLine*) – Source line from which to map.
- **trigger_destination** (*constants.OutputTriggerLine*) – Destination line to which to map.
- **mode** (*None, optional*) – Always None for this version of the VISA specification.

Returns Return value of the library call.

Return type *StatusCode*

memory_allocation(*session: NewType.<locals>.new_type*, *size: int*, *extended: bool = False*) → Tuple[int, pyvisa.constants.StatusCode]

Allocate memory from a resource's memory region.

Corresponds to viMemAlloc* functions of the VISA library.

Parameters

- **session** (*VISASession*) – Unique logical identifier to a session.
- **size** (*int*) – Specifies the size of the allocation.
- **extended** (*bool, optional*) – Use 64 bits offset independent of the platform.

Returns

- *int* – offset of the allocated memory
- *StatusCode* – Return value of the library call.

memory_free(*session: NewType.<locals>.new_type*, *offset: int*, *extended: bool = False*) → pyvisa.constants.StatusCode

Frees memory previously allocated using the memory_allocation() operation.

Corresponds to viMemFree* function of the VISA library.

Parameters

- **session** (*VISASession*) – Unique logical identifier to a session.
- **offset** (*int*) – Offset of the memory to free.
- **extended** (*bool, optional*) – Use 64 bits offset independent of the platform.

Returns Return value of the library call.

Return type *StatusCode*

move(*session: NewType.<locals>.new_type*, *source_space: pyvisa.constants.AddressSpace*, *source_offset: int*, *source_width: pyvisa.constants.DataWidth*, *destination_space: pyvisa.constants.AddressSpace*, *destination_offset: int*, *destination_width: pyvisa.constants.DataWidth*, *length: int*) → pyvisa.constants.StatusCode

Moves a block of data.

Corresponds to viMove function of the VISA library.

Parameters

- **session** (*VISASession*) – Unique logical identifier to a session.
- **source_space** (*constants.AddressSpace*) – Specifies the address space of the source.
- **source_offset** (*int*) – Offset of the starting address or register from which to read.
- **source_width** (*constants.DataWidth*) – Specifies the data width of the source.
- **destination_space** (*constants.AddressSpace*) – Specifies the address space of the destination.
- **destination_offset** (*int*) – Offset of the starting address or register to which to write.
- **destination_width** (*constants.DataWidth*) – Specifies the data width of the destination.
- **length** (*int*) – Number of elements to transfer, where the data width of the elements to transfer is identical to the source data width.

Returns Return value of the library call.

Return type *StatusCode*

move_asynchronously(*session: NewType.<locals>.new_type*, *source_space: pyvisa.constants.AddressSpace*, *source_offset: int*, *source_width: pyvisa.constants.DataWidth*, *destination_space: pyvisa.constants.AddressSpace*, *destination_offset: int*, *destination_width: pyvisa.constants.DataWidth*, *length: int*) → Tuple[NewType.<locals>.new_type, pyvisa.constants.StatusCode]

Moves a block of data asynchronously.

Corresponds to viMoveAsync function of the VISA library.

Parameters

- **session** (*VISASession*) – Unique logical identifier to a session.
- **source_space** (*constants.AddressSpace*) – Specifies the address space of the source.
- **source_offset** (*int*) – Offset of the starting address or register from which to read.
- **source_width** (*constants.DataWidth*) – Specifies the data width of the source.
- **destination_space** (*constants.AddressSpace*) – Specifies the address space of the destination.
- **destination_offset** (*int*) – Offset of the starting address or register to which to write.
- **destination_width** (*constants.DataWidth*) – Specifies the data width of the destination.
- **length** (*int*) – Number of elements to transfer, where the data width of the elements to transfer is identical to the source data width.

Returns

- *VISAJobID* – Job identifier of this asynchronous move operation
- *StatusCode* – Return value of the library call.

move_in(*session: NewType.<locals>.new_type, space: pyvisa.constants.AddressSpace, offset: int, length: int, width: Union[typing_extensions.Literal[8, 16, 32, 64][8, 16, 32, 64], pyvisa.constants.DataWidth], extended: bool = False*) → Tuple[List[int], pyvisa.constants.StatusCode]

Move a block of data to local memory from the given address space and offset.

Corresponds to viMoveIn* functions of the VISA library.

Parameters

- **session** (*VISASession*) – Unique logical identifier to a session.
- **space** (*constants.AddressSpace*) – Address space from which to move the data.
- **offset** (*int*) – Offset (in bytes) of the address or register from which to read.
- **length** (*int*) – Number of elements to transfer, where the data width of the elements to transfer is identical to the source data width.
- **width** (*Union[Literal[8, 16, 32, 64], constants.DataWidth]*) – Number of bits to read per element.
- **extended** (*bool, optional*) – Use 64 bits offset independent of the platform, by default False.

Returns

- **data** (*List[int]*) – Data read from the bus
- **status_code** (*StatusCode*) – Return value of the library call.

Raises `ValueError` – Raised if an invalid width is specified.

move_in_16(*session: NewType.<locals>.new_type*, *space: pyvisa.constants.AddressSpace*, *offset: int*, *length: int*, *extended: bool = False*) → Tuple[List[int], pyvisa.constants.StatusCode]

Moves an 16-bit block of data to local memory.

Corresponds to viMoveIn816 functions of the VISA library.

Parameters

- **session** (*VISASession*) – Unique logical identifier to a session.
- **space** (*constants.AddressSpace*) – Address space from which to move the data.
- **offset** (*int*) – Offset (in bytes) of the address or register from which to read.
- **length** (*int*) – Number of elements to transfer, where the data width of the elements to transfer is identical to the source data width.
- **extended** (*bool, optional*) – Use 64 bits offset independent of the platform, by default False.

Returns

- **data** (*List[int]*) – Data read from the bus
- **status_code** (*StatusCode*) – Return value of the library call.

move_in_32(*session: NewType.<locals>.new_type*, *space: pyvisa.constants.AddressSpace*, *offset: int*, *length: int*, *extended: bool = False*) → Tuple[List[T]]

Moves an 32-bit block of data to local memory.

Corresponds to viMoveIn32* functions of the VISA library.

Parameters

- **session** (*VISASession*) – Unique logical identifier to a session.
- **space** (*constants.AddressSpace*) – Address space from which to move the data.
- **offset** (*int*) – Offset (in bytes) of the address or register from which to read.
- **length** (*int*) – Number of elements to transfer, where the data width of the elements to transfer is identical to the source data width.
- **extended** (*bool, optional*) – Use 64 bits offset independent of the platform, by default False.

Returns

- **data** (*List[int]*) – Data read from the bus
- **status_code** (*StatusCode*) – Return value of the library call.

move_in_64(*session: NewType.<locals>.new_type*, *space: pyvisa.constants.AddressSpace*, *offset: int*, *length: int*, *extended: bool = False*) → Tuple[List[int], pyvisa.constants.StatusCode]

Moves an 64-bit block of data to local memory.

Corresponds to viMoveIn8* functions of the VISA library.

Parameters

- **session** (*VISASession*) – Unique logical identifier to a session.
- **space** (*constants.AddressSpace*) – Address space from which to move the data.

- **offset** (*int*) – Offset (in bytes) of the address or register from which to read.
- **length** (*int*) – Number of elements to transfer, where the data width of the elements to transfer is identical to the source data width.
- **extended** (*bool, optional*) – Use 64 bits offset independent of the platform, by default False.

Returns

- **data** (*List[int]*) – Data read from the bus
- **status_code** (*StatusCode*) – Return value of the library call.

move_in_8(*session: NewType.<locals>.new_type*, *space: pyvisa.constants.AddressSpace*, *offset: int*, *length: int*, *extended: bool = False*) → Tuple[List[int], pyvisa.constants.StatusCode]

Moves an 8-bit block of data to local memory.

Corresponds to viMoveIn8* functions of the VISA library.

Parameters

- **session** (*VISASession*) – Unique logical identifier to a session.
- **space** (*constants.AddressSpace*) – Address space from which to move the data.
- **offset** (*int*) – Offset (in bytes) of the address or register from which to read.
- **length** (*int*) – Number of elements to transfer, where the data width of the elements to transfer is identical to the source data width.
- **extended** (*bool, optional*) – Use 64 bits offset independent of the platform, by default False.

Returns

- **data** (*List[int]*) – Data read from the bus
- **status_code** (*StatusCode*) – Return value of the library call.

move_out(*session: NewType.<locals>.new_type, space: pyvisa.constants.AddressSpace, offset: int, length: int, data: Iterable[int], width: Union[typing_extensions.Literal[8, 16, 32, 64][8, 16, 32, 64], pyvisa.constants.DataWidth], extended: bool = False*) → pyvisa.constants.StatusCode

Move a block of data from local memory to the given address space and offset.

Corresponds to viMoveOut* functions of the VISA library.

Parameters

- **session** (*VISASession*) – Unique logical identifier to a session.
- **space** (*constants.AddressSpace*) – Address space into which move the data.
- **offset** (*int*) – Offset (in bytes) of the address or register from which to read.
- **length** (*int*) – Number of elements to transfer, where the data width of the elements to transfer is identical to the source data width.
- **data** (*Iterable[int]*) – Data to write to bus.
- **width** (*Union[Literal[8, 16, 32, 64], constants.DataWidth]*) – Number of bits to per element.
- **extended** (*bool, optional*) – Use 64 bits offset independent of the platform, by default False.

Returns Return value of the library call.

Return type *StatusCode*

Raises `ValueError` – Raised if an invalid width is specified.

move_out_16(*session: NewType.<locals>.new_type, space: pyvisa.constants.AddressSpace, offset: int, length: int, data: Iterable[int], extended: bool = False*) → pyvisa.constants.StatusCode

Moves an 16-bit block of data from local memory.

Corresponds to viMoveOut16* functions of the VISA library.

Parameters

- **session** (*VISASession*) – Unique logical identifier to a session.
- **space** (*constants.AddressSpace*) – Address space into which move the data.
- **offset** (*int*) – Offset (in bytes) of the address or register from which to read.
- **length** (*int*) – Number of elements to transfer, where the data width of the elements to transfer is identical to the source data width.
- **data** (*Iterable[int]*) – Data to write to bus.
- **extended** (*bool, optional*) – Use 64 bits offset independent of the platform, by default False.

Returns Return value of the library call.

Return type *StatusCode*

move_out_32(*session: NewType.<locals>.new_type, space: pyvisa.constants.AddressSpace, offset: int, length: int, data: Iterable[int], extended: bool = False*) → pyvisa.constants.StatusCode

Moves an 32-bit block of data from local memory.

Corresponds to viMoveOut32* functions of the VISA library.

Parameters

- **session** (*VISASession*) – Unique logical identifier to a session.
- **space** (*constants.AddressSpace*) – Address space into which move the data.
- **offset** (*int*) – Offset (in bytes) of the address or register from which to read.
- **length** (*int*) – Number of elements to transfer, where the data width of the elements to transfer is identical to the source data width.
- **data** (*Iterable[int]*) – Data to write to bus.
- **extended** (*bool, optional*) – Use 64 bits offset independent of the platform, by default False.

Returns Return value of the library call.

Return type *StatusCode*

move_out_64(*session: NewType.<locals>.new_type, space: pyvisa.constants.AddressSpace, offset: int, length: int, data: Iterable[int], extended: bool = False*) → pyvisa.constants.StatusCode

Moves an 64-bit block of data from local memory.

Corresponds to viMoveOut64* functions of the VISA library.

Parameters

- **session** (*VISASession*) – Unique logical identifier to a session.

- **space** (*constants.AddressSpace*) – Address space into which move the data.
- **offset** (*int*) – Offset (in bytes) of the address or register from which to read.
- **length** (*int*) – Number of elements to transfer, where the data width of the elements to transfer is identical to the source data width.
- **data** (*Iterable[int]*) – Data to write to bus.
- **extended** (*bool, optional*) – Use 64 bits offset independent of the platform, by default False.

Returns Return value of the library call.

Return type *StatusCode*

move_out_8 (*session: NewType.<locals>.new_type, space: pyvisa.constants.AddressSpace, offset: int, length: int, data: Iterable[int], extended: bool = False*) → pyvisa.constants.StatusCode

Moves an 8-bit block of data from local memory.

Corresponds to viMoveOut8* functions of the VISA library.

Parameters

- **session** (*VISASession*) – Unique logical identifier to a session.
- **space** (*constants.AddressSpace*) – Address space into which move the data.
- **offset** (*int*) – Offset (in bytes) of the address or register from which to read.
- **length** (*int*) – Number of elements to transfer, where the data width of the elements to transfer is identical to the source data width.
- **data** (*Iterable[int]*) – Data to write to bus.
- **extended** (*bool, optional*) – Use 64 bits offset independent of the platform, by default False.

Returns Return value of the library call.

Return type *StatusCode*

open (*session: NewType.<locals>.new_type, resource_name: str, access_mode: pyvisa.constants.AccessModes = <AccessModes.no_lock: 0>, open_timeout: int = 0*) → Tuple[NewType.<locals>.new_type, pyvisa.constants.StatusCode]

Opens a session to the specified resource.

Corresponds to viOpen function of the VISA library.

Parameters

- **session** (*VISARMSession*) – Resource Manager session (should always be a session returned from open_default_resource_manager()).
- **resource_name** (*str*) – Unique symbolic name of a resource.
- **access_mode** (*constants.AccessModes, optional*) – Specifies the mode by which the resource is to be accessed.
- **open_timeout** (*int*) – If the access_mode parameter requests a lock, then this parameter specifies the absolute time period (in milliseconds) that the resource waits to get unlocked before this operation returns an error.

Returns

- *VISASession* – Unique logical identifier reference to a session
- *StatusCode* – Return value of the library call.

open_default_resource_manager() → Tuple[NewType.<locals>.new_type, pyvisa.constants.StatusCode]

This function returns a session to the Default Resource Manager resource.

Corresponds to viOpenDefaultRM function of the VISA library.

Returns

- *VISARMSession* – Unique logical identifier to a Default Resource Manager session
- *StatusCode* – Return value of the library call.

out_16(*session: NewType.<locals>.new_type*, *space: pyvisa.constants.AddressSpace*, *offset: int*, *data: int*, *extended: bool = False*) → pyvisa.constants.StatusCode

Write a 16-bit value to the specified memory space and offset.

Corresponds to viOut16* functions of the VISA library.

Parameters

- **session** (*VISASession*) – Unique logical identifier to a session.
- **space** (*constants.AddressSpace*) – Address space into which to write.
- **offset** (*int*) – Offset (in bytes) of the address or register from which to read.
- **data** (*int*) – Data to write to bus.
- **extended** (*bool, optional*) – Use 64 bits offset independent of the platform.

Returns Return value of the library call.

Return type *StatusCode*

out_32(*session: NewType.<locals>.new_type, space: pyvisa.constants.AddressSpace, offset: int, data: Iterable[int], extended: bool = False*) → pyvisa.constants.StatusCode

Write a 32-bit value to the specified memory space and offset.

Corresponds to viOut32* functions of the VISA library.

Parameters

- **session** (*VISASession*) – Unique logical identifier to a session.
- **space** (*constants.AddressSpace*) – Address space into which to write.
- **offset** (*int*) – Offset (in bytes) of the address or register from which to read.
- **data** (*int*) – Data to write to bus.
- **extended** (*bool, optional*) – Use 64 bits offset independent of the platform.

Returns Return value of the library call.

Return type *StatusCode*

out_64(*session: NewType.<locals>.new_type, space: pyvisa.constants.AddressSpace, offset: int, data: Iterable[int], extended: bool = False*) → pyvisa.constants.StatusCode

Write a 64-bit value to the specified memory space and offset.

Corresponds to viOut64* functions of the VISA library.

Parameters

- **session** (*VISASession*) – Unique logical identifier to a session.
- **space** (*constants.AddressSpace*) – Address space into which to write.
- **offset** (*int*) – Offset (in bytes) of the address or register from which to read.

- **data** (*int*) – Data to write to bus.
- **extended** (*bool, optional*) – Use 64 bits offset independent of the platform.

Returns Return value of the library call.

Return type *StatusCode*

out_8(*session: NewType.<locals>.new_type*, *space: pyvisa.constants.AddressSpace*, *offset: int*, *data: int*, *extended: bool = False*) → pyvisa.constants.StatusCode

Write an 8-bit value to the specified memory space and offset.

Corresponds to viOut8* functions of the VISA library.

Parameters

- **session** (*VISASession*) – Unique logical identifier to a session.
- **space** (*constants.AddressSpace*) – Address space into which to write.
- **offset** (*int*) – Offset (in bytes) of the address or register from which to read.
- **data** (*int*) – Data to write to bus.
- **extended** (*bool, optional*) – Use 64 bits offset independent of the platform.

Returns Return value of the library call.

Return type *StatusCode*

parse_resource(*session: NewType.<locals>.new_type*, *resource_name: str*) → Tuple[pyvisa.highlevel.ResourceInfo, pyvisa.constants.StatusCode]

Parse a resource string to get the interface information.

Corresponds to viParseRsrc function of the VISA library.

Parameters

- **session** (*VISARMSession*) – Resource Manager session (should always be the Default Resource Manager for VISA returned from open_default_resource_manager()).
- **resource_name** (*str*) – Unique symbolic name of a resource.

Returns

- *ResourceInfo* – Resource information with interface type and board number
- *StatusCode* – Return value of the library call.

parse_resource_extended(*session: NewType.<locals>.new_type*, *resource_name: str*) → Tuple[pyvisa.highlevel.ResourceInfo, pyvisa.constants.StatusCode]

Parse a resource string to get extended interface information.

Corresponds to viParseRsrcEx function of the VISA library.

Parameters

- **session** (*VISARMSession*) – Resource Manager session (should always be the Default Resource Manager for VISA returned from open_default_resource_manager()).
- **resource_name** (*str*) – Unique symbolic name of a resource.

Returns

- *ResourceInfo* – Resource information with interface type and board number
- *StatusCode* – Return value of the library call.

peek(*session: NewType.<locals>.new_type, address: NewType.<locals>.new_type, width: Union[typing_extensions.Literal[8, 16, 32, 64][8, 16, 32, 64], pyvisa.constants.DataWidth]*) → Tuple[int, pyvisa.constants.StatusCode]

Read an 8, 16, 32, or 64-bit value from the specified address.

Corresponds to viPeek* functions of the VISA library.

Parameters

- **session** (*VISASession*) – Unique logical identifier to a session.
- **address** (*VISAMemoryAddress*) – Source address to read the value.
- **width** (*Union[Literal[8, 16, 32, 64], constants.DataWidth]*) – Number of bits to read.

Returns

- **data** (*int*) – Data read from bus
- **status_code** (*StatusCode*) – Return value of the library call.

Raises ValueError – Raised if an invalid width is specified.

peek_16(*session: NewType.<locals>.new_type*, *address: NewType.<locals>.new_type*) → Tuple[int, pyvisa.constants.StatusCode]

Read an 16-bit value from the specified address.

Corresponds to viPeek16 function of the VISA library.

Parameters

- **session** (*VISASession*) – Unique logical identifier to a session.
- **address** (*VISAMemoryAddress*) – Source address to read the value.

Returns

- *int* – Data read from bus
- *StatusCode* – Return value of the library call.

peek_32(*session: NewType.<locals>.new_type*, *address: NewType.<locals>.new_type*) → Tuple[int, pyvisa.constants.StatusCode]

Read an 32-bit value from the specified address.

Corresponds to viPeek32 function of the VISA library.

Parameters

- **session** (*VISASession*) – Unique logical identifier to a session.
- **address** (*VISAMemoryAddress*) – Source address to read the value.

Returns

- *int* – Data read from bus
- *StatusCode* – Return value of the library call.

peek_64(*session: NewType.<locals>.new_type*, *address: NewType.<locals>.new_type*) → Tuple[int, pyvisa.constants.StatusCode]

Read an 64-bit value from the specified address.

Corresponds to viPeek64 function of the VISA library.

Parameters

- **session** (*VISASession*) – Unique logical identifier to a session.

- **address** (*VISAMemoryAddress*) – Source address to read the value.

Returns

- *int* – Data read from bus
- *StatusCode* – Return value of the library call.

peek_8(*session: NewType.<locals>.new_type*, *address: NewType.<locals>.new_type*) → Tuple[int, pyvisa.constants.StatusCode]

Read an 8-bit value from the specified address.

Corresponds to viPeek8 function of the VISA library.

Parameters

- **session** (*VISASession*) – Unique logical identifier to a session.
- **address** (*VISAMemoryAddress*) – Source address to read the value.

Returns

- *int* – Data read from bus
- *StatusCode* – Return value of the library call.

poke(*session: NewType.<locals>.new_type, address: NewType.<locals>.new_type, width: Union[typing_extensions.Literal[8, 16, 32, 64][8, 16, 32, 64], pyvisa.constants.DataWidth], data: int*) → pyvisa.constants.StatusCode

Writes an 8, 16, 32, or 64-bit value from the specified address.

Corresponds to viPoke* functions of the VISA library.

Parameters

- **session** (*VISASession*) – Unique logical identifier to a session.
- **address** (*VISAMemoryAddress*) – Source address to read the value.
- **width** (*Union[Literal[8, 16, 32, 64], constants.DataWidth]*) – Number of bits to read.
- **data** (*int*) – Data to write to the bus

Returns **status_code** – Return value of the library call.

Return type *StatusCode*

Raises `ValueError` – Raised if an invalid width is specified.

poke_16(*session: NewType.<locals>.new_type*, *address: NewType.<locals>.new_type*, *data: int*) → pyvisa.constants.StatusCode

Write an 16-bit value to the specified address.

Corresponds to viPoke16 function of the VISA library.

Parameters

- **session** (*VISASession*) – Unique logical identifier to a session.
- **address** (*VISAMemoryAddress*) – Source address to read the value.
- **data** (*int*) – Data to write.

Returns Return value of the library call.

Return type *StatusCode*

poke_32 (*session: NewType.<locals>.new_type*, *address: NewType.<locals>.new_type*, *data: int*) → pyvisa.constants.StatusCode

Write an 32-bit value to the specified address.

Corresponds to viPoke32 function of the VISA library.

Parameters

- **session** (*VISASession*) – Unique logical identifier to a session.
- **address** (*VISAMemoryAddress*) – Source address to read the value.
- **data** (*int*) – Data to write.

Returns Return value of the library call.

Return type *StatusCode*

poke_64 (*session: NewType.<locals>.new_type*, *address: NewType.<locals>.new_type*, *data: int*) → pyvisa.constants.StatusCode

Write an 64-bit value to the specified address.

Corresponds to viPoke64 function of the VISA library.

Parameters

- **session** (*VISASession*) – Unique logical identifier to a session.
- **address** (*VISAMemoryAddress*) – Source address to read the value.
- **data** (*int*) – Data to write.

Returns Return value of the library call.

Return type *StatusCode*

poke_8 (*session: NewType.<locals>.new_type*, *address: NewType.<locals>.new_type*, *data: int*) → pyvisa.constants.StatusCode

Write an 8-bit value to the specified address.

Corresponds to viPoke8 function of the VISA library.

Parameters

- **session** (*VISASession*) – Unique logical identifier to a session.
- **address** (*VISAMemoryAddress*) – Source address to read the value.
- **data** (*int*) – Data to write.

Returns Return value of the library call.

Return type *StatusCode*

read (*session: NewType.<locals>.new_type*, *count: int*) → Tuple[bytes, pyvisa.constants.StatusCode]

Reads data from device or interface synchronously.

Corresponds to viRead function of the VISA library.

Parameters

- **session** (*VISASession*) – Unique logical identifier to a session.
- **count** (*int*) – Number of bytes to be read.

Returns

- *bytes* – Date read
- *StatusCode* – Return value of the library call.

read_asynchronously(*session: NewType.<locals>.new_type*, *count: int*) → Tuple[SupportsBytes, NewType.<locals>.new_type, pyvisa.constants.StatusCode]

Reads data from device or interface asynchronously.

Corresponds to viReadAsync function of the VISA library. Since the asynchronous operation may complete before the function call return implementation should make sure that get_buffer_from_id will be able to return the proper buffer before this method returns.

Parameters

- **session** (*VISASession*) – Unique logical identifier to a session.
- **count** (*int*) – Number of bytes to be read.

Returns

- *SupportsBytes* – Buffer that will be filled during the asynchronous operation.
- *VISAJobID* – Id of the asynchronous job
- *StatusCode* – Return value of the library call.

read_memory(*session: NewType.<locals>.new_type, space: pyvisa.constants.AddressSpace, offset: int, width: Union[typing_extensions.Literal[8, 16, 32, 64][8, 16, 32, 64], pyvisa.constants.DataWidth], extended: bool = False*) → Tuple[int, pyvisa.constants.StatusCode]

Read a value from the specified memory space and offset.

Corresponds to viIn* functions of the VISA library.

Parameters

- **session** (*VISASession*) – Unique logical identifier to a session.
- **space** (*constants.AddressSpace*) – Specifies the address space from which to read.
- **offset** (*int*) – Offset (in bytes) of the address or register from which to read.
- **width** (*Union[Literal[8, 16, 32, 64], constants.DataWidth]*) – Number of bits to read (8, 16, 32 or 64).
- **extended** (*bool, optional*) – Use 64 bits offset independent of the platform.

Returns

- **data** (*int*) – Data read from memory
- **status_code** (*StatusCode*) – Return value of the library call.

Raises ValueError – Raised if an invalid width is specified.

read_stb(*session: NewType.<locals>.new_type*) → Tuple[int, pyvisa.constants.StatusCode]

Reads a status byte of the service request.

Corresponds to viReadSTB function of the VISA library.

Parameters **session** (*VISASession*) – Unique logical identifier to a session.

Returns

- *int* – Service request status byte
- *StatusCode* – Return value of the library call.

read_to_file(*session: NewType.<locals>.new_type*, *filename: str*, *count: int*) → Tuple[int, pyvisa.constants.StatusCode]

Read data synchronously, and store the transferred data in a file.

Corresponds to viReadToFile function of the VISA library.

Parameters

- **session** (*VISASession*) – Unique logical identifier to a session.
- **filename** (*str*) – Name of file to which data will be written.
- **count** (*int*) – Number of bytes to be read.

Returns

- *int* – Number of bytes actually transferred
- *StatusCode* – Return value of the library call.

resource_manager = None

Default ResourceManager instance for this library.

set_attribute(*session: NewType.<locals>.new_type*, *attribute: pyvisa.constants.ResourceAttribute*, *attribute_state: Any*) → pyvisa.constants.StatusCode

Set the state of an attribute.

Corresponds to viSetAttribute function of the VISA library.

Parameters

- **session** (*VISASession*) – Unique logical identifier to a session.
- **attribute** (*constants.ResourceAttribute*) – Attribute for which the state is to be modified.
- **attribute_state** (*Any*) – The state of the attribute to be set for the specified object.

Returns Return value of the library call.

Return type *StatusCode*

set_buffer(*session: NewType.<locals>.new_type*, *mask: pyvisa.constants.BufferType*, *size: int*) → pyvisa.constants.StatusCode

Set the size for the formatted I/O and/or low-level I/O communication buffer(s).

Corresponds to viSetBuf function of the VISA library.

Parameters

- **session** (*VISASession*) – Unique logical identifier to a session.
- **mask** (*constants.BufferType*) – Specifies the type of buffer.
- **size** (*int*) – The size to be set for the specified buffer(s).

Returns Return value of the library call.

Return type *StatusCode*

status_description(*session: NewType.<locals>.new_type*, *status: pyvisa.constants.StatusCode*) → Tuple[str, pyvisa.constants.StatusCode]

Return a user-readable description of the status code passed to the operation.

Corresponds to viStatusDesc function of the VISA library.

Parameters

- **session** (*VISASession*) – Unique logical identifier to a session.
- **status** (StatusCode) – Status code to interpret.

Returns

- *str* – User-readable string interpretation of the status code.
- *StatusCode* – Return value of the library call.

terminate(*session: NewType.<locals>.new_type*, *degree: None*, *job_id: NewType.<locals>.new_type*) → pyvisa.constants.StatusCode

Request a VISA session to terminate normal execution of an operation.

Corresponds to viTerminate function of the VISA library.

Parameters

- **session** (*VISASession*) – Unique logical identifier to a session.
- **degree** (*None*) – Not used in this version of the VISA specification.
- **job_id** (*VISAJobId*) – Specifies an operation identifier. If a user passes None as the job_id value to viTerminate(), a VISA implementation should abort any calls in the current process executing on the specified vi. Any call that is terminated this way should return VI_ERROR_ABORT.

Returns Return value of the library call.

Return type *StatusCode*

uninstall_all_visa_handlers(*session: Optional[NewType.<locals>.new_type]*) → None

Uninstalls all previously installed handlers for a particular session.

Parameters **session** (*VISASession | None*) – Unique logical identifier to a session. If None, operates on all sessions.

uninstall_handler(*session: NewType.<locals>.new_type, event_type: pyvisa.constants.EventType, handler: Callable[[NewType.<locals>.new_type, pyvisa.constants.EventType, NewType.<locals>.new_type, Any], None], user_handle: Any = None*) → pyvisa.constants.StatusCode

Uninstall handlers for events.

Corresponds to viUninstallHandler function of the VISA library.

Parameters

- **session** (*VISASession*) – Unique logical identifier to a session.
- **event_type** (*constants.EventType*) – Logical event identifier.
- **handler** (*VISAHandler*) – Handler to be uninstalled by a client application.
- **user_handle** – A value specified by an application that can be used for identifying handlers uniquely in a session for an event. The modified value of the user_handle as returned by install_handler should be used instead of the original value.

Returns Return value of the library call.

Return type *StatusCode*

uninstall_visa_handler(*session: NewType.<locals>.new_type, event_type: pyvisa.constants.EventType, handler: Callable[[NewType.<locals>.new_type, pyvisa.constants.EventType, NewType.<locals>.new_type, Any], None], user_handle: Any = None*) → None

Uninstalls handlers for events.

Parameters

- **session** (*VISASession*) – Unique logical identifier to a session.

- **event_type** (*constants.EventType*) – Logical event identifier.
- **handler** (*VISAHandler*) – Handler to be uninstalled by a client application.
- **user_handle** – The user handle returned by install_visa_handler.

unlock (*session: NewType.<locals>.new_type*) → pyvisa.constants.StatusCode
Relinquish a lock for the specified resource.

Corresponds to viUnlock function of the VISA library.

Parameters **session** (*VISASession*) – Unique logical identifier to a session.

Returns Return value of the library call.

Return type *StatusCode*

unmap_address (*session: NewType.<locals>.new_type*) → pyvisa.constants.StatusCode
Unmap memory space previously mapped by map_address().

Corresponds to viUnmapAddress function of the VISA library.

Parameters **session** (*VISASession*) – Unique logical identifier to a session.

Returns Return value of the library call.

Return type *StatusCode*

unmap_trigger (*session: NewType.<locals>.new_type*, *trigger_source: pyvisa.constants.InputTriggerLine*, *trigger_destination: pyvisa.constants.OutputTriggerLine*) → pyvisa.constants.StatusCode
Undo a previous map between a trigger source line and a destination line.

Corresponds to viUnmapTrigger function of the VISA library.

Parameters

- **session** (*VISASession*) – Unique logical identifier to a session.
- **trigger_source** (*constants.InputTriggerLine*) – Source line used in previous map.
- **trigger_destination** (*constants.OutputTriggerLine*) – Destination line used in previous map.

Returns Return value of the library call.

Return type *StatusCode*

usb_control_in (*session: NewType.<locals>.new_type*, *request_type_bitmap_field: int*, *request_id: int*, *request_value: int*, *index: int*, *length: int = 0*) → Tuple[bytes, pyvisa.constants.StatusCode]
Perform a USB control pipe transfer from the device.

Corresponds to viUsbControlIn function of the VISA library.

Parameters

- **session** (*VISASession*) – Unique logical identifier to a session.
- **request_type_bitmap_field** (*int*) – bmRequestType parameter of the setup stage of a USB control transfer.
- **request_id** (*int*) – bRequest parameter of the setup stage of a USB control transfer.
- **request_value** (*int*) – wValue parameter of the setup stage of a USB control transfer.

- **index** (*int*) – wIndex parameter of the setup stage of a USB control transfer. This is usually the index of the interface or endpoint.
- **length** (*int, optional*) – wLength parameter of the setup stage of a USB control transfer. This value also specifies the size of the data buffer to receive the data from the optional data stage of the control transfer.

Returns

- *bytes* – The data buffer that receives the data from the optional data stage of the control transfer
- *StatusCode* – Return value of the library call.

usb_control_out(*session: NewType.<locals>.new_type*, *request_type_bitmap_field: int*, *request_id: int*, *request_value: int*, *index: int*, *data: bytes = b''*) → pyvisa.constants.StatusCode

Perform a USB control pipe transfer to the device.

Corresponds to viUsbControlOut function of the VISA library.

Parameters

- **session** (*VISASession*) – Unique logical identifier to a session.
- **request_type_bitmap_field** (*int*) – bmRequestType parameter of the setup stage of a USB control transfer.
- **request_id** (*int*) – bRequest parameter of the setup stage of a USB control transfer.
- **request_value** (*int*) – wValue parameter of the setup stage of a USB control transfer.
- **index** (*int*) – wIndex parameter of the setup stage of a USB control transfer. This is usually the index of the interface or endpoint.
- **data** (*bytes, optional*) – The data buffer that sends the data in the optional data stage of the control transfer.

Returns Return value of the library call.

Return type *StatusCode*

vxi_command_query(*session: NewType.<locals>.new_type*, *mode: pyvisa.constants.VXICommands*, *command: int*) → Tuple[int, pyvisa.constants.StatusCode]

Send the device a miscellaneous command or query and/or retrieves the response to a previous query.

Corresponds to viVxiCommandQuery function of the VISA library.

Parameters

- **session** (*VISASession*) – Unique logical identifier to a session.
- **mode** (*constants.VXICommands*) – Specifies whether to issue a command and/or retrieve a response.
- **command** (*int*) – The miscellaneous command to send.

Returns

- *int* – The response retrieved from the device
- *StatusCode* – Return value of the library call.

wait_on_event(*session: NewType.<locals>.new_type*, *in_event_type: pyvisa.constants.EventType*, *timeout: int*) → Tuple[pyvisa.constants.EventType, NewType.<locals>.new_type, pyvisa.constants.StatusCode]

Wait for an occurrence of the specified event for a given session.

Corresponds to viWaitOnEvent function of the VISA library.

Parameters

- **session** (*VISASession*) – Unique logical identifier to a session.
- **in_event_type** (*constants.EventType*) – Logical identifier of the event(s) to wait for.
- **timeout** (*int*) – Absolute time period in time units that the resource shall wait for a specified event to occur before returning the time elapsed error. The time unit is in milliseconds.

Returns

- *constants.EventType* – Logical identifier of the event actually received
- *VISAEventContext* – A handle specifying the unique occurrence of an event
- *StatusCode* – Return value of the library call.

write(*session: NewType.<locals>.new_type*, *data: bytes*) → Tuple[int, pyvisa.constants.StatusCode]

Write data to device or interface synchronously.

Corresponds to viWrite function of the VISA library.

Parameters

- **session** (*VISASession*) – Unique logical identifier to a session.
- **data** (*bytes*) – Data to be written.

Returns

- *int* – Number of bytes actually transferred
- *StatusCode* – Return value of the library call.

write_asynchronously(*session: NewType.<locals>.new_type*, *data: bytes*) → Tuple[NewType.<locals>.new_type, pyvisa.constants.StatusCode]

Write data to device or interface asynchronously.

Corresponds to viWriteAsync function of the VISA library.

Parameters

- **session** (*VISASession*) – Unique logical identifier to a session.
- **data** (*bytes*) – Data to be written.

Returns

- *VISAJobID* – Job ID of this asynchronous write operation
- *StatusCode* – Return value of the library call.

write_from_file(*session: NewType.<locals>.new_type*, *filename: str*, *count: int*) → Tuple[int, pyvisa.constants.StatusCode]

Take data from a file and write it out synchronously.

Corresponds to viWriteFromFile function of the VISA library.

Parameters

- **session** (*VISASession*) – Unique logical identifier to a session.
- **filename** (*str*) – Name of file from which data will be read.
- **count** (*int*) – Number of bytes to be written.

Returns

- *int* – Number of bytes actually transferred
- *StatusCode* – Return value of the library call.

write_memory(*session: NewType.<locals>.new_type, space: pyvisa.constants.AddressSpace, offset: int, data: int, width: Union[typing_extensions.Literal[8, 16, 32, 64][8, 16, 32, 64], pyvisa.constants.DataWidth], extended: bool = False*) → pyvisa.constants.StatusCode

Write a value to the specified memory space and offset.

Parameters

- **session** (*VISASession*) – Unique logical identifier to a session.
- **space** (*constants.AddressSpace*) – Specifies the address space.
- **offset** (*int*) – Offset (in bytes) of the address or register from which to read.
- **data** (*int*) – Data to write to bus.
- **width** (*Union[Literal[8, 16, 32, 64], constants.DataWidth]*) – Number of bits to read.
- **extended** (*bool, optional*) – Use 64 bits offset independent of the platform, by default False.

Returns Return value of the library call.

Return type *StatusCode*

Raises ValueError – Raised if an invalid width is specified.

1.4.2 Resource Manager

class pyvisa.highlevel.**ResourceInfo**(*interface_type*, *interface_board_number*, *resource_class*, *resource_name*, *alias*)

Resource extended information

Named tuple with information about a resource. Returned by some *ResourceManager* methods.

Interface_type Interface type of the given resource string. *pyvisa.constants.InterfaceType*

Interface_board_number Board number of the interface of the given resource string. We allow None since serial resources may not sometimes be easily described by a single number in particular on Linux system.

Resource_class Specifies the resource class (for example, "INSTR") of the given resource string.

Resource_name This is the expanded version of the given resource string. The format should be similar to the VISA-defined canonical resource name.

Alias Specifies the user-defined alias for the given resource string.

class pyvisa.highlevel.**ResourceManager**

VISA Resource Manager.

close() → None
Close the resource manager session.

last_status
Last status code returned for an operation with this Resource Manager.

list_resources(*query: str = '?*::INSTR'*) → Tuple[str, ...]
Return a tuple of all connected devices matching query.

Notes

The query uses the VISA Resource Regular Expression syntax - which is not the same as the Python regular expression syntax. (see below)

The VISA Resource Regular Expression syntax is defined in the VISA Library specification: http://www.ivifoundation.org/docs/vpp43.pdf

Symbol Meaning ———- ———-

? Matches any one character.

Makes the character that follows it an ordinary character instead of special character. For example, when a question mark follows a backslash (?), it matches the ? character instead of any one character.

[list] Matches any one character from the enclosed list. You can use a hyphen to match a range of characters.

[^list] Matches any character not in the enclosed list. You can use a hyphen to match a range of characters.

- Matches 0 or more occurrences of the preceding character or expression.
- Matches 1 or more occurrences of the preceding character or expression.

Exp|exp Matches either the preceding or following expression. The or operator | matches the entire expression that precedes or follows it and not just the character that precedes or follows it. For example, VXI|GPIB means (VXI)|(GPIB), not VX(I|G)PIB.

(exp) Grouping characters or expressions.

Thus the default query, '?*::INSTR', matches any sequences of characters ending ending with '::INSTR'.

On some platforms, devices that are already open are not returned.

list_resources_info(*query: str = '?*::INSTR'*) → Dict[str, pyvisa.highlevel.ResourceInfo]
Get extended information about all connected devices matching query.

For details of the VISA Resource Regular Expression syntax used in query, refer to list_resources().

Returns Mapping of resource name to ResourceInfo

Return type Dict[str, *ResourceInfo*]

open_bare_resource(*resource_name: str*, *access_mode: pyvisa.constants.AccessModes = <AccessModes.no_lock: 0>*, *open_timeout: int = 0*) → Tuple[NewType.<locals>.new_type, pyvisa.constants.StatusCode]
Open the specified resource without wrapping into a class.

Parameters

- **resource_name** (*str*) – Name or alias of the resource to open.

- **access_mode** (*constants.AccessModes, optional*) – Specifies the mode by which the resource is to be accessed, by default constants.AccessModes.no_lock
- **open_timeout** (*int, optional*) – If the access_mode parameter requests a lock, then this parameter specifies the absolute time period (in milliseconds) that the resource waits to get unlocked before this operation returns an error, by default constants.VI_TMO_IMMEDIATE.

Returns

- *VISASession* – Unique logical identifier reference to a session.
- *StatusCode* – Return value of the library call.

open_resource(*resource_name: str, access_mode: pyvisa.constants.AccessModes = <AccessModes.no_lock: 0>, open_timeout: int = 0, resource_pyclass: Optional[Type[Resource]] = None, **kwargs*) → Resource

Return an instrument for the resource name.

Parameters

- **resource_name** (*str*) – Name or alias of the resource to open.
- **access_mode** (*constants.AccessModes, optional*) – Specifies the mode by which the resource is to be accessed, by default constants.AccessModes.no_lock
- **open_timeout** (*int, optional*) – If the access_mode parameter requests a lock, then this parameter specifies the absolute time period (in milliseconds) that the resource waits to get unlocked before this operation returns an error, by default constants.VI_TMO_IMMEDIATE.
- **resource_pyclass** (*Optional[Type[Resource]], optional*) – Resource Python class to use to instantiate the Resource. Defaults to None: select based on the resource name.
- **kwargs** (*Any*) – Keyword arguments to be used to change instrument attributes after construction.

Returns Subclass of Resource matching the resource.

Return type *Resource*

resource_info(*resource_name: str, extended: bool = True*) → pyvisa.highlevel.ResourceInfo

Get the (extended) information of a particular resource.

Parameters

- **resource_name** (*str*) – Unique symbolic name of a resource.
- **extended** (*bool, optional*) – Also get extended information (ie. resource_class, resource_name, alias)

session

Resource Manager session handle.

Raises errors.InvalidSession – Raised if the session is closed.

1.4.3 Resource classes

Resources are high level abstractions to managing specific sessions. An instance of one of these classes is returned by the *open_resource()* depending on the resource type.

Generic classes

- *Resource*
- *MessageBasedResource*
- *RegisterBasedResource*

Specific Classes

- *SerialInstrument*
- *TCPIPInstrument*
- *TCPIPSocket*
- *USBInstrument*
- *USBRaw*
- *GPIBInstrument*
- *GPIBInterface*
- *FirewireInstrument*
- *PXIInstrument*
- *PXIInstrument*
- *VXIInstrument*
- *VXIMemory*
- *VXIBackplane*

class pyvisa.resources.**Resource**(*resource_manager: pyvisa.highlevel.ResourceManager*, *resource_name: str*)

Base class for resources.

Do not instantiate directly, use *pyvisa.highlevel.ResourceManager.open_resource()*.

before_close() → None
Called just before closing an instrument.

clear() → None
Clear this resource.

close() → None
Closes the VISA session and marks the handle as invalid.

disable_event(*event_type: pyvisa.constants.EventType*, *mechanism: pyvisa.constants.EventMechanism*) → None
Disable notification for an event type(s) via the specified mechanism(s).

Parameters

- **event_type** (*constants.EventType*) – Logical event identifier.
- **mechanism** (*constants.EventMechanism*) – Specifies event handling mechanisms to be disabled.

discard_events(*event_type: pyvisa.constants.EventType*, *mechanism: pyvisa.constants.EventMechanism*) → None
Discards event occurrences for an event type and mechanism in this resource.

Parameters

- **event_type** (*constants.EventType*) – Logical event identifier.
- **mechanism** (*constants.EventMechanism*) – Specifies event handling mechanisms to be disabled.

enable_event(*event_type: pyvisa.constants.EventType*, *mechanism: pyvisa.constants.EventMechanism*, *context: None = None*) → None
Enable event occurrences for specified event types and mechanisms in this resource.

Parameters

- **event_type** (*constants.EventType*) – Logical event identifier.
- **mechanism** (*constants.EventMechanism*) – Specifies event handling mechanisms to be enabled
- **context** (*None*) – Not currently used, leave as None.

get_visa_attribute(*name: pyvisa.constants.ResourceAttribute*) → Any
Retrieves the state of an attribute in this resource.

One should prefer the dedicated descriptor for often used attributes since those perform checks and automatic conversion on the value.

Parameters **name** (*constants.ResourceAttribute*) – Resource attribute for which the state query is made.

Returns The state of the queried attribute for a specified resource.

Return type Any

ignore_warning(**warnings_constants*) → AbstractContextManager[T_co]
Ignoring warnings context manager for the current resource.

Parameters **warnings_constants** (constants.StatusCode) – Constants identifying the warnings to ignore.

implementation_version
Resource version that identifies the revisions or implementations of a resource.

install_handler(*event_type: pyvisa.constants.EventType*, *handler: Callable[[NewType.<locals>.new_type, pyvisa.constants.EventType, NewType.<locals>.new_type, Any], None]*, *user_handle=None*) → Any
Install handlers for event callbacks in this resource.

Parameters

- **event_type** (*constants.EventType*) – Logical event identifier.
- **handler** (*VISAHandler*) – Handler function to be installed by a client application.
- **user_handle** – A value specified by an application that can be used for identifying handlers uniquely for an event type. Depending on the backend they may be restriction on the possible values. Look at the backend *install_visa_handler* for more details.

Returns User handle in a format amenable to the backend. This is this representation of the handle that should be used when unistalling a handler.

Return type Any

interface_number
Board number for the given interface.

interface_type
Interface type of the given session.

last_status
Last status code for this session.

lock(*timeout: Union[float, typing_extensions.Literal['default']][default]] = 'default', requested_key: Optional[str] = None*) → str
Establish a shared lock to the resource.

Parameters

- **timeout** (*Union[float, Literal["default"]], optional*) – Absolute time period (in milliseconds) that a resource waits to get unlocked by the locking session before returning an error. Defaults to "default" which means use self.timeout.
- **requested_key** (*Optional[str], optional*) – Access key used by another session with which you want your session to share a lock or None to generate a new shared access key.

Returns A new shared access key if requested_key is None, otherwise, same value as the requested_key

Return type str

lock_context(*timeout: Union[float, typing_extensions.Literal['default']][default]] = 'default', requested_key: Optional[str] = 'exclusive'*) → Iterator[Optional[str]]
A context that locks

Parameters

- **timeout** (*Union[float, Literal["default"]], optional*) – Absolute time period (in milliseconds) that a resource waits to get unlocked by the locking session before returning an error. Defaults to "default" which means use self.timeout.
- **requested_key** (*Optional[str], optional*) – When using default of 'exclusive' the lock is an exclusive lock. Otherwise it is the access key for the shared lock or None to generate a new shared access key.

Yields *Optional[str]* – The access_key if applicable.

lock_excl(*timeout: Union[float, typing_extensions.Literal['default']][default]] = 'default'*) → None
Establish an exclusive lock to the resource.

Parameters **timeout** (*Union[float, Literal["default"]], optional*) – Absolute time period (in milliseconds) that a resource waits to get unlocked by the locking session before returning an error. Defaults to "default" which means use self.timeout.

lock_state
Current locking state of the resource.

open(*access_mode: pyvisa.constants.AccessModes = <AccessModes.no_lock: 0>, open_timeout: int = 5000*) → None
Opens a session to the specified resource.

Parameters

- **access_mode** (*constants.AccessModes, optional*) – Specifies the mode by which the resource is to be accessed. Defaults to constants.AccessModes.no_lock.
- **open_timeout** (*int, optional*) – If the access_mode parameter requests a lock, then this parameter specifies the absolute time period (in milliseconds) that the resource waits to get unlocked before this operation returns an error. Defaults to 5000.

classmethod register(*interface_type: pyvisa.constants.InterfaceType*, *resource_class: str*) → Callable[[Type[T]], Type[T]]

Create a decorator to register a class.

The class is associated to an interface type, resource class pair.

Parameters

- **interface_type** (constants.InterfaceType) – Interface type for which to register a wrapper class.
- **resource_class** (*str*) – Resource class for which to register a wrapper class.

Returns Decorator registering the class. Raises TypeError if some VISA attributes are missing on the registered class.

Return type Callable[[Type[T]], Type[T]]

resource_class

Resource class (for example, "INSTR") as defined by the canonical resource name.

resource_info

Get the extended information of this resource.

resource_manager = None

Reference to the resource manager used by this resource

resource_manufacturer_name

Manufacturer name of the vendor that implemented the VISA library.

resource_name

Unique identifier for a resource compliant with the address structure.

session

Resource session handle.

Raises errors.InvalidSession – Raised if session is closed.

set_visa_attribute(*name: pyvisa.constants.ResourceAttribute*, *state: Any*) → pyvisa.constants.StatusCode

Set the state of an attribute.

One should prefer the dedicated descriptor for often used attributes since those perform checks and automatic conversion on the value.

Parameters

- **name** (*constants.ResourceAttribute*) – Attribute for which the state is to be modified.
- **state** (*Any*) – The state of the attribute to be set for the specified object.

Returns Return value of the library call.

Return type *constants.StatusCode*

spec_version

Version of the VISA specification to which the implementation is compliant.

timeout

Timeout in milliseconds for all resource I/O operations.

uninstall_handler(*event_type: pyvisa.constants.EventType*, *handler: Callable[[NewType.<locals>.new_type, pyvisa.constants.EventType, NewType.<locals>.new_type, Any], None]*, *user_handle=None*) → None

Uninstalls handlers for events in this resource.

Parameters

- **event_type** (*constants.EventType*) – Logical event identifier.
- **handler** (*VISAHandler*) – Handler function to be uninstalled by a client application.
- **user_handle** (*Any*) – The user handle returned by install_handler.

unlock() → None
Relinquishes a lock for the specified resource.

visa_attributes_classes = {<class 'pyvisa.attributes.AttrVI_ATTR_RSRC_NAME'>, <class '
VISA attribute descriptor classes that can be used to introspect the supported attributes and the possible values. The "often used" ones are generally directly available on the resource.

visalib = None
Reference to the VISA library instance used by the resource

wait_on_event(*in_event_type: pyvisa.constants.EventType*, *timeout: int*, *capture_timeout: bool = False*) → pyvisa.resources.resource.WaitResponse
Waits for an occurrence of the specified event in this resource.

in_event_type [constants.EventType] Logical identifier of the event(s) to wait for.

timeout [int] Absolute time period in time units that the resource shall wait for a specified event to occur before returning the time elapsed error. The time unit is in milliseconds. None means waiting forever if necessary.

capture_timeout [bool, optional] When True will not produce a VisaIOError(VI_ERROR_TMO) but instead return a WaitResponse with timed_out=True.

Returns Object that contains event_type, context and ret value.

Return type WaitResponse

wrap_handler(*callable: Callable[[Resource, pyvisa.events.Event, Any], None]*) → Callable[[NewType.<locals>.new_type, pyvisa.constants.EventType, NewType.<locals>.new_type, Any], None]
Wrap an event handler to provide the signature expected by VISA.

The handler is expected to have the following signature: handler(resource: Resource, event: Event, user_handle: Any) -> None.

The wrapped handler should be used only to handle events on the resource used to wrap the handler.

class pyvisa.resources.**MessageBasedResource**(*resource_manager: pyvisa.highlevel.ResourceManager*, *resource_name: str*)
Base class for resources that use message based communication.

CR = '\r'

LF = '\n'

allow_dma
Should I/O accesses use DMA (True) or Programmed I/O (False).

assert_trigger() → None
Sends a software trigger to the device.

before_close() → None
Called just before closing an instrument.

chunk_size = 20480
Number of bytes to read at a time. Some resources (serial) may not support large chunk sizes.

clear() → None
Clear this resource.

close() → None
Closes the VISA session and marks the handle as invalid.

disable_event(*event_type: pyvisa.constants.EventType*, *mechanism: pyvisa.constants.EventMechanism*) → None
Disable notification for an event type(s) via the specified mechanism(s).

Parameters

- **event_type** (*constants.EventType*) – Logical event identifier.
- **mechanism** (*constants.EventMechanism*) – Specifies event handling mechanisms to be disabled.

discard_events(*event_type: pyvisa.constants.EventType*, *mechanism: pyvisa.constants.EventMechanism*) → None
Discards event occurrences for an event type and mechanism in this resource.

Parameters

- **event_type** (*constants.EventType*) – Logical event identifier.
- **mechanism** (*constants.EventMechanism*) – Specifies event handling mechanisms to be disabled.

enable_event(*event_type: pyvisa.constants.EventType*, *mechanism: pyvisa.constants.EventMechanism*, *context: None = None*) → None
Enable event occurrences for specified event types and mechanisms in this resource.

Parameters

- **event_type** (*constants.EventType*) – Logical event identifier.
- **mechanism** (*constants.EventMechanism*) – Specifies event handling mechanisms to be enabled
- **context** (*None*) – Not currently used, leave as None.

encoding
Encoding used for read and write operations.

flush(*mask: pyvisa.constants.BufferOperation*) → None
Manually clears the specified buffers.

Depending on the value of the mask this can cause the buffer data to be written to the device.

Parameters **mask** (*constants.BufferOperation*) – Specifies the action to be taken with flushing the buffer. See highlevel.VisaLibraryBase.flush for a detailed description.

get_visa_attribute(*name: pyvisa.constants.ResourceAttribute*) → Any
Retrieves the state of an attribute in this resource.

One should prefer the dedicated descriptor for often used attributes since those perform checks and automatic conversion on the value.

Parameters **name** (*constants.ResourceAttribute*) – Resource attribute for which the state query is made.

Returns The state of the queried attribute for a specified resource.

Return type Any

ignore_warning(**warnings_constants*) → AbstractContextManager[T_co]
Ignoring warnings context manager for the current resource.

Parameters **warnings_constants** (constants.StatusCode) – Constants identifying the warnings to ignore.

implementation_version
Resource version that identifies the revisions or implementations of a resource.

This attribute value is defined by the individual manufacturer and increments with each new revision. The format of the value has the upper 12 bits as the major number of the version, the next lower 12 bits as the minor number of the version, and the lowest 8 bits as the sub-minor number of the version.

VISA Attribute VI_ATTR_RSRC_IMPL_VERSION (1073676291)

Type int

Range 0 <= value <= 4294967295

install_handler(*event_type: pyvisa.constants.EventType, handler: Callable[[NewType.<locals>.new_type, pyvisa.constants.EventType, NewType.<locals>.new_type, Any], None], user_handle=None*) → Any
Install handlers for event callbacks in this resource.

Parameters

- **event_type** (*constants.EventType*) – Logical event identifier.
- **handler** (*VISAHandler*) – Handler function to be installed by a client application.
- **user_handle** – A value specified by an application that can be used for identifying handlers uniquely for an event type. Depending on the backend they may be restriction on the possible values. Look at the backend *install_visa_handler* for more details.

Returns User handle in a format amenable to the backend. This is this representation of the handle that should be used when unistalling a handler.

Return type Any

interface_number
Board number for the given interface. :VISA Attribute: VI_ATTR_INTF_NUM (1073676662) :type: int :range: 0 <= value <= 65535

interface_type
Interface type of the given session. :VISA Attribute: VI_ATTR_INTF_TYPE (1073676657) :type: :class:pyvisa.constants.InterfaceType

io_protocol
IO protocol to use. See the attribute definition for more details.

last_status
Last status code for this session.

lock(*timeout: Union[float, typing_extensions.Literal['default']][default]] = 'default', requested_key: Optional[str] = None*) → str
Establish a shared lock to the resource.

Parameters

- **timeout** (*Union[float, Literal["default"]], optional*) – Absolute time period (in milliseconds) that a resource waits to get unlocked by the locking session before returning an error. Defaults to "default" which means use self.timeout.
- **requested_key** (*Optional[str], optional*) – Access key used by another session with which you want your session to share a lock or None to generate a new shared access key.

Returns A new shared access key if requested_key is None, otherwise, same value as the requested_key

Return type str

lock_context(*timeout: Union[float, typing_extensions.Literal['default']][default]] = 'default', requested_key: Optional[str] = 'exclusive'*) → Iterator[Optional[str]]

A context that locks

Parameters

- **timeout** (*Union[float, Literal["default"]], optional*) – Absolute time period (in milliseconds) that a resource waits to get unlocked by the locking session before returning an error. Defaults to "default" which means use self.timeout.
- **requested_key** (*Optional[str], optional*) – When using default of 'exclusive' the lock is an exclusive lock. Otherwise it is the access key for the shared lock or None to generate a new shared access key.

Yields *Optional[str]* – The access_key if applicable.

lock_excl(*timeout: Union[float, typing_extensions.Literal['default']][default]] = 'default'*) → None

Establish an exclusive lock to the resource.

Parameters **timeout** (*Union[float, Literal["default"]], optional*) – Absolute time period (in milliseconds) that a resource waits to get unlocked by the locking session before returning an error. Defaults to "default" which means use self.timeout.

lock_state

Current locking state of the resource.

The resource can be unlocked, locked with an exclusive lock, or locked with a shared lock.

VISA Attribute VI_ATTR_RSRC_LOCK_STATE (1073676292)

Type :class:pyvisa.constants.AccessModes

open(*access_mode: pyvisa.constants.AccessModes = <AccessModes.no_lock: 0>, open_timeout: int = 5000*) → None

Opens a session to the specified resource.

Parameters

- **access_mode** (*constants.AccessModes, optional*) – Specifies the mode by which the resource is to be accessed. Defaults to constants.AccessModes.no_lock.
- **open_timeout** (*int, optional*) – If the access_mode parameter requests a lock, then this parameter specifies the absolute time period (in milliseconds) that the resource waits to get unlocked before this operation returns an error. Defaults to 5000.

query(*message: str, delay: Optional[float] = None*) → str

A combination of write(message) and read()

Parameters

- **message** (*str*) – The message to send.

- **delay** (*Optional[float], optional*) – Delay in seconds between write and read operations. If None, defaults to self.query_delay.

Returns Answer from the device.

Return type str

query_ascii_values(*message: str, converter: Union[typing_extensions.Literal['s', 'b', 'c', 'd', 'o', 'x', 'X', 'e', 'E', 'f', 'F', 'g', 'G'][s, b, c, d, o, x, X, e, E, f, F, g, G], Callable[[str], Any]] = 'f', separator: Union[str, Callable[[str], Iterable[str]]] = ',', container: Union[Type[CT_co], Callable[[Iterable[T_co]], Sequence[T_co]]] = <class 'list'>, delay: Optional[float] = None*) → Sequence[Any]

Query the device for values in ascii format returning an iterable of values.

Parameters

- **message** (*str*) – The message to send.
- **converter** (*ASCII_CONVERTER, optional*) – Str format of function to convert each value. Default to "f".
- **separator** (*Union[str, Callable[[str], Iterable[str]]]*) – str or callable used to split the data into individual elements. If a str is given, data.split(separator) is used. Default to ",".
- **container** (*Union[Type, Callable[[Iterable], Sequence]], optional*) – Container type to use for the output data. Possible values are: list, tuple, np.ndarray, etc, Default to list.
- **delay** (*Optional[float], optional*) – Delay in seconds between write and read operations. If None, defaults to self.query_delay.

Returns Parsed data.

Return type Sequence

query_binary_values(*message: str, datatype: typing_extensions.Literal['s', 'b', 'B', 'h', 'H', 'i', 'I', 'l', 'L', 'q', 'Q', 'f', 'd'][s, b, B, h, H, i, I, l, L, q, Q, f, d] = 'f', is_big_endian: bool = False, container: Union[Type[CT_co], Callable[[Iterable[T_co]], Sequence[T_co]]] = <class 'list'>, delay: Optional[float] = None, header_fmt: typing_extensions.Literal['ieee', 'hp', 'empty'][ieee, hp, empty] = 'ieee', expect_termination: bool = True, data_points: int = 0, chunk_size: Optional[int] = None*) → Sequence[Union[int, float]]

Query the device for values in binary format returning an iterable of values.

Parameters

- **message** (*str*) – The message to send.
- **datatype** (*BINARY_DATATYPES, optional*) – Format string for a single element. See struct module. 'f' by default.
- **is_big_endian** (*bool, optional*) – Are the data in big or little endian order. Defaults to False.
- **container** (*Union[Type, Callable[[Iterable], Sequence]], optional*) – Container type to use for the output data. Possible values are: list, tuple, np.ndarray, etc, Default to list.
- **delay** (*Optional[float], optional*) – Delay in seconds between write and read operations. If None, defaults to self.query_delay.

- **header_fmt** (*util.BINARY_HEADERS, optional*) – Format of the header prefixing the data. Defaults to 'ieee'.
- **expect_termination** (*bool, optional*) – When set to False, the expected length of the binary values block does not account for the final termination character (the read termination). Defaults to True.
- **data_points** (*int, optional*) – Number of points expected in the block. This is used only if the instrument does not report it itself. This will be converted in a number of bytes based on the datatype. Defaults to 0.
- **chunk_size** (*int, optional*) – Size of the chunks to read from the device. Using larger chunks may be faster for large amount of data.

Returns Data read from the device.

Return type Sequence[Union[int, float]]

query_delay = 0.0
Delay in s to sleep between the write and read occuring in a query

read(*termination: Optional[str] = None*, *encoding: Optional[str] = None*) → str
Read a string from the device.

Reading stops when the device stops sending (e.g. by setting appropriate bus lines), or the termination characters sequence was detected. Attention: Only the last character of the termination characters is really used to stop reading, however, the whole sequence is compared to the ending of the read string message. If they don't match, a warning is issued.

Parameters

- **termination** (*Optional[str], optional*) – Alternative character termination to use. If None, the value of write_termination is used. Defaults to None.
- **encoding** (*Optional[str], optional*) – Alternative encoding to use to turn bytes into str. If None, the value of encoding is used. Defaults to None.

Returns Message read from the instrument and decoded.

Return type str

read_ascii_values(*converter: Union[typing_extensions.Literal['s', 'b', 'c', 'd', 'o', 'x', 'X', 'e', 'E', 'f', 'F', 'g', 'G'][s, b, c, d, o, x, X, e, E, f, F, g, G], Callable[[str], Any]] = 'f', separator: Union[str, Callable[[str], Iterable[str]]] = ',', container: Union[Type[CT_co], Callable[[Iterable[T_co]], Sequence[T_co]]] = <class 'list'>*) → Sequence[T_co]
Read values from the device in ascii format returning an iterable of values.

Parameters **converter** (*ASCII_CONVERTER, optional*) – Str format of function to convert each value. Default to "f".

separator [Union[str, Callable[[str], Iterable[str]]]] str or callable used to split the data into individual elements. If a str is given, data.split(separator) is used. Default to ",".

container [Union[Type, Callable[[Iterable], Sequence]], optional] Container type to use for the output data. Possible values are: list, tuple, np.ndarray, etc, Default to list.

Returns Parsed data.

Return type Sequence

read_binary_values(*datatype: typing_extensions.Literal['s', 'b', 'B', 'h', 'H', 'i', 'I', 'l', 'L', 'q', 'Q', 'f', 'd'][s, b, B, h, H, i, I, l, L, q, Q, f, d] = 'f', is_big_endian: bool = False, container: Union[Type[CT_co], Callable[[Iterable[T_co]], Sequence[T_co]]] = <class 'list'>, header_fmt: typing_extensions.Literal['ieee', 'hp', 'empty'][ieee, hp, empty] = 'ieee', expect_termination: bool = True, data_points: int = 0, chunk_size: Optional[int] = None*) → Sequence[Union[int, float]]

Read values from the device in binary format returning an iterable of values.

Parameters

- **datatype** (*BINARY_DATATYPES, optional*) – Format string for a single element. See struct module. 'f' by default.
- **is_big_endian** (*bool, optional*) – Are the data in big or little endian order. Defaults to False.
- **container** (*Union[Type, Callable[[Iterable], Sequence]], optional*) – Container type to use for the output data. Possible values are: list, tuple, np.ndarray, etc, Default to list.
- **header_fmt** (*util.BINARY_HEADERS, optional*) – Format of the header prefixing the data. Defaults to 'ieee'.
- **expect_termination** (*bool, optional*) – When set to False, the expected length of the binary values block does not account for the final termination character (the read termination). Defaults to True.
- **data_points** (*int, optional*) – Number of points expected in the block. This is used only if the instrument does not report it itself. This will be converted in a number of bytes based on the datatype. Defaults to 0.
- **chunk_size** (*int, optional*) – Size of the chunks to read from the device. Using larger chunks may be faster for large amount of data.

Returns Data read from the device.

Return type Sequence[Union[int, float]]

read_bytes(*count: int*, *chunk_size: Optional[int] = None*, *break_on_termchar: bool = False*) → bytes

Read a certain number of bytes from the instrument.

Parameters

- **count** (*int*) – The number of bytes to read from the instrument.
- **chunk_size** (*Optional[int], optional*) – The chunk size to use to perform the reading. If count > chunk_size multiple low level operations will be performed. Defaults to None, meaning the resource wide set value is set.
- **break_on_termchar** (*bool, optional*) – Should the reading stop when a termination character is encountered or when the message ends. Defaults to False.

Returns Bytes read from the instrument.

Return type bytes

read_raw(*size: Optional[int] = None*) → bytes

Read the unmodified string sent from the instrument to the computer.

In contrast to read(), no termination characters are stripped.

Parameters **size** (*Optional[int], optional*) – The chunk size to use to perform the reading. Defaults to None, meaning the resource wide set value is set.

Returns Bytes read from the instrument.

Return type bytes

read_stb() → int
Service request status register.

read_termination
Read termination character.

read_termination_context(*new_termination: str*) → Iterator[T_co]

classmethod register(*interface_type: pyvisa.constants.InterfaceType*, *resource_class: str*) → Callable[[Type[T]], Type[T]]
Create a decorator to register a class.

The class is associated to an interface type, resource class pair.

Parameters

- **interface_type** (constants.InterfaceType) – Interface type for which to register a wrapper class.
- **resource_class** (*str*) – Resource class for which to register a wrapper class.

Returns Decorator registering the class. Raises TypeError if some VISA attributes are missing on the registered class.

Return type Callable[[Type[T]], Type[T]]

resource_class
Resource class as defined by the canonical resource name.

Possible values are: INSTR, INTFC, SOCKET, RAW...

VISA Attribute VI_ATTR_RSRC_CLASS (3221159937)

resource_info
Get the extended information of this resource.

resource_manufacturer_name
Manufacturer name of the vendor that implemented the VISA library.

This attribute is not related to the device manufacturer attributes.

Note The value of this attribute is for display purposes only and not for programmatic decisions, as the value can differ between VISA implementations and/or revisions.

VISA Attribute VI_ATTR_RSRC_MANF_NAME (3221160308)

resource_name
Unique identifier for a resource compliant with the address structure. :VISA Attribute: VI_ATTR_RSRC_NAME (3221159938)

send_end
Should END be asserted during the transfer of the last byte of the buffer.

session
Resource session handle.

Raises errors.InvalidSession – Raised if session is closed.

set_visa_attribute(*name: pyvisa.constants.ResourceAttribute, state: Any*) → pyvisa.constants.StatusCode

Set the state of an attribute.

One should prefer the dedicated descriptor for often used attributes since those perform checks and automatic conversion on the value.

Parameters

- **name** (*constants.ResourceAttribute*) – Attribute for which the state is to be modified.
- **state** (*Any*) – The state of the attribute to be set for the specified object.

Returns Return value of the library call.

Return type *constants.StatusCode*

spec_version

Version of the VISA specification to which the implementation is compliant.

The format of the value has the upper 12 bits as the major number of the version, the next lower 12 bits as the minor number of the version, and the lowest 8 bits as the sub-minor number of the version. The current VISA specification defines the value to be 00300000h.

VISA Attribute VI_ATTR_RSRC_SPEC_VERSION (1073676656)

Type int

Range 0 <= value <= 4294967295

stb

Service request status register.

timeout

Timeout in milliseconds for all resource I/O operations.

This value is used when accessing the device associated with the given session.

Special values:

- **immediate (VI_TMO_IMMEDIATE): 0** (for convenience, any value smaller than 1 is considered as 0)
- **infinite (VI_TMO_INFINITE): float('+inf')** (for convenience, None is considered as `float('+inf')`)

To set an **infinite** timeout, you can also use:

```
>>> del instrument.timeout
```

A timeout value of VI_TMO_IMMEDIATE means that operations should never wait for the device to respond. A timeout value of VI_TMO_INFINITE disables the timeout mechanism.

VISA Attribute VI_ATTR_TMO_VALUE (1073676314)

Type int

Range 0 <= value <= 4294967295

uninstall_handler(*event_type: pyvisa.constants.EventType, handler: Callable[[NewType.<locals>.new_type, pyvisa.constants.EventType, NewType.<locals>.new_type, Any], None], user_handle=None*) → None

Uninstalls handlers for events in this resource.

Parameters

- **event_type** (*constants.EventType*) – Logical event identifier.
- **handler** (*VISAHandler*) – Handler function to be uninstalled by a client application.
- **user_handle** (*Any*) – The user handle returned by install_handler.

unlock() → None
Relinquishes a lock for the specified resource.

visa_attributes_classes = {<class 'pyvisa.attributes.AttrVI_ATTR_RSRC_NAME'>, <class '

wait_on_event(*in_event_type: pyvisa.constants.EventType*, *timeout: int*, *capture_timeout: bool = False*) → pyvisa.resources.resource.WaitResponse
Waits for an occurrence of the specified event in this resource.

in_event_type [constants.EventType] Logical identifier of the event(s) to wait for.

timeout [int] Absolute time period in time units that the resource shall wait for a specified event to occur before returning the time elapsed error. The time unit is in milliseconds. None means waiting forever if necessary.

capture_timeout [bool, optional] When True will not produce a VisaIOError(VI_ERROR_TMO) but instead return a WaitResponse with timed_out=True.

Returns Object that contains event_type, context and ret value.

Return type WaitResponse

wrap_handler(*callable: Callable[[Resource, pyvisa.events.Event, Any], None]*) → Callable[[NewType.<locals>.new_type, pyvisa.constants.EventType, NewType.<locals>.new_type, Any], None]
Wrap an event handler to provide the signature expected by VISA.

The handler is expected to have the following signature: handler(resource: Resource, event: Event, user_handle: Any) -> None.

The wrapped handler should be used only to handle events on the resource used to wrap the handler.

write(*message: str*, *termination: Optional[str] = None*, *encoding: Optional[str] = None*) → int
Write a string message to the device.

The write_termination is always appended to it.

Parameters

- **message** (*str*) – The message to be sent.
- **termination** (*Optional[str], optional*) – Alternative character termination to use. If None, the value of write_termination is used. Defaults to None.
- **encoding** (*Optional[str], optional*) – Alternative encoding to use to turn str into bytes. If None, the value of encoding is used. Defaults to None.

Returns Number of bytes written.

Return type int

write_ascii_values(*message: str, values: Sequence[Any], converter: Union[typing_extensions.Literal['s', 'b', 'c', 'd', 'o', 'x', 'X', 'e', 'E', 'f', 'F', 'g', 'G'][s, b, c, d, o, x, X, e, E, f, F, g, G], Callable[[str], Any]] = 'f', separator: Union[str, Callable[[Iterable[str]], str]] = ',', termination: Optional[str] = None, encoding: Optional[str] = None*)
Write a string message to the device followed by values in ascii format.

The write_termination is always appended to it.

Parameters

- **message** (*str*) – Header of the message to be sent.
- **values** (*Sequence[Any]*) – Data to be writen to the device.
- **converter** (*Union[str, Callable[[Any], str]], optional*) – Str formatting codes or function used to convert each value. Defaults to "f".
- **separator** (*Union[str, Callable[[Iterable[str]], str]], optional*) – Str or callable that join the values in a single str. If a str is given, separator.join(values) is used. Defaults to ','
- **termination** (*Optional[str], optional*) – Alternative character termination to use. If None, the value of write_termination is used. Defaults to None.
- **encoding** (*Optional[str], optional*) – Alternative encoding to use to turn str into bytes. If None, the value of encoding is used. Defaults to None.

Returns Number of bytes written.

Return type int

write_binary_values(*message: str, values: Sequence[Any], datatype: typing_extensions.Literal['s', 'b', 'B', 'h', 'H', 'i', 'I', 'l', 'L', 'q', 'Q', 'f', 'd'][s, b, B, h, H, i, I, l, L, q, Q, f, d] = 'f', is_big_endian: bool = False, termination: Optional[str] = None, encoding: Optional[str] = None, header_fmt: typing_extensions.Literal['ieee', 'hp', 'empty'][ieee, hp, empty] = 'ieee'*)

Write a string message to the device followed by values in binary format.

The write_termination is always appended to it.

Parameters

- **message** (*str*) – The header of the message to be sent.
- **values** (*Sequence[Any]*) – Data to be written to the device.
- **datatype** (*util.BINARY_DATATYPES, optional*) – The format string for a single element. See struct module.
- **is_big_endian** (*bool, optional*) – Are the data in big or little endian order.
- **termination** (*Optional[str], optional*) – Alternative character termination to use. If None, the value of write_termination is used. Defaults to None.
- **encoding** (*Optional[str], optional*) – Alternative encoding to use to turn str into bytes. If None, the value of encoding is used. Defaults to None.
- **header_fmt** (*util.BINARY_HEADERS*) – Format of the header prefixing the data.

Returns Number of bytes written.

Return type int

write_raw(*message: bytes*) → int

Write a byte message to the device.

Parameters **message** (*bytes*) – The message to be sent.

Returns Number of bytes written

Return type int

write_termination
Write termination character.

class pyvisa.resources.**RegisterBasedResource**(*resource_manager: pyvisa.highlevel.ResourceManager*, *resource_name: str*)
Base class for resources that use register based communication.

before_close() → None
Called just before closing an instrument.

clear() → None
Clear this resource.

close() → None
Closes the VISA session and marks the handle as invalid.

disable_event(*event_type: pyvisa.constants.EventType*, *mechanism: pyvisa.constants.EventMechanism*) → None
Disable notification for an event type(s) via the specified mechanism(s).

Parameters

- **event_type** (*constants.EventType*) – Logical event identifier.
- **mechanism** (*constants.EventMechanism*) – Specifies event handling mechanisms to be disabled.

discard_events(*event_type: pyvisa.constants.EventType*, *mechanism: pyvisa.constants.EventMechanism*) → None
Discards event occurrences for an event type and mechanism in this resource.

Parameters

- **event_type** (*constants.EventType*) – Logical event identifier.
- **mechanism** (*constants.EventMechanism*) – Specifies event handling mechanisms to be disabled.

enable_event(*event_type: pyvisa.constants.EventType*, *mechanism: pyvisa.constants.EventMechanism*, *context: None = None*) → None
Enable event occurrences for specified event types and mechanisms in this resource.

Parameters

- **event_type** (*constants.EventType*) – Logical event identifier.
- **mechanism** (*constants.EventMechanism*) – Specifies event handling mechanisms to be enabled
- **context** (*None*) – Not currently used, leave as None.

get_visa_attribute(*name: pyvisa.constants.ResourceAttribute*) → Any
Retrieves the state of an attribute in this resource.

One should prefer the dedicated descriptor for often used attributes since those perform checks and automatic conversion on the value.

Parameters **name** (*constants.ResourceAttribute*) – Resource attribute for which the state query is made.

Returns The state of the queried attribute for a specified resource.

Return type Any

ignore_warning(**warnings_constants*) → AbstractContextManager[T_co]
Ignoring warnings context manager for the current resource.

Parameters **warnings_constants** (constants.StatusCode) – Constants identifying the warnings to ignore.

implementation_version
Resource version that identifies the revisions or implementations of a resource.

This attribute value is defined by the individual manufacturer and increments with each new revision. The format of the value has the upper 12 bits as the major number of the version, the next lower 12 bits as the minor number of the version, and the lowest 8 bits as the sub-minor number of the version.

VISA Attribute VI_ATTR_RSRC_IMPL_VERSION (1073676291)

Type int

Range 0 <= value <= 4294967295

install_handler(*event_type: pyvisa.constants.EventType, handler: Callable[[NewType.<locals>.new_type, pyvisa.constants.EventType, NewType.<locals>.new_type, Any], None], user_handle=None*) → Any
Install handlers for event callbacks in this resource.

Parameters

- **event_type** (*constants.EventType*) – Logical event identifier.
- **handler** (*VISAHandler*) – Handler function to be installed by a client application.
- **user_handle** – A value specified by an application that can be used for identifying handlers uniquely for an event type. Depending on the backend they may be restriction on the possible values. Look at the backend *install_visa_handler* for more details.

Returns User handle in a format amenable to the backend. This is this representation of the handle that should be used when unistalling a handler.

Return type Any

interface_number
Board number for the given interface. :VISA Attribute: VI_ATTR_INTF_NUM (1073676662) :type: int :range: 0 <= value <= 65535

interface_type
Interface type of the given session. :VISA Attribute: VI_ATTR_INTF_TYPE (1073676657) :type: :class:pyvisa.constants.InterfaceType

last_status
Last status code for this session.

lock(*timeout: Union[float, typing_extensions.Literal['default'][default]] = 'default', requested_key: Optional[str] = None*) → str
Establish a shared lock to the resource.

Parameters

- **timeout** (*Union[float, Literal["default"]], optional*) – Absolute time period (in milliseconds) that a resource waits to get unlocked by the locking session before returning an error. Defaults to "default" which means use self.timeout.

- **requested_key** (*Optional[str], optional*) – Access key used by another session with which you want your session to share a lock or None to generate a new shared access key.

Returns A new shared access key if requested_key is None, otherwise, same value as the requested_key

Return type str

lock_context(*timeout: Union[float, typing_extensions.Literal['default'][default]] = 'default', requested_key: Optional[str] = 'exclusive'*) → Iterator[Optional[str]]

A context that locks

Parameters

- **timeout** (*Union[float, Literal["default"]], optional*) – Absolute time period (in milliseconds) that a resource waits to get unlocked by the locking session before returning an error. Defaults to "default" which means use self.timeout.
- **requested_key** (*Optional[str], optional*) – When using default of 'exclusive' the lock is an exclusive lock. Otherwise it is the access key for the shared lock or None to generate a new shared access key.

Yields *Optional[str]* – The access_key if applicable.

lock_excl(*timeout: Union[float, typing_extensions.Literal['default'][default]] = 'default'*) → None

Establish an exclusive lock to the resource.

Parameters **timeout** (*Union[float, Literal["default"]], optional*) – Absolute time period (in milliseconds) that a resource waits to get unlocked by the locking session before returning an error. Defaults to "default" which means use self.timeout.

lock_state

Current locking state of the resource.

The resource can be unlocked, locked with an exclusive lock, or locked with a shared lock.

VISA Attribute VI_ATTR_RSRC_LOCK_STATE (1073676292)

Type :class:pyvisa.constants.AccessModes

move_in(*space: pyvisa.constants.AddressSpace, offset: int, length: int, width: pyvisa.constants.DataWidth, extended: bool = False*) → List[int]

Move a block of data to local memory from the given address space and offset.

Corresponds to viMoveIn* functions of the VISA library.

Parameters

- **space** (*constants.AddressSpace*) – Address space from which to move the data.
- **offset** (*int*) – Offset (in bytes) of the address or register from which to read.
- **length** (*int*) – Number of elements to transfer, where the data width of the elements to transfer is identical to the source data width.
- **width** (*Union[Literal[8, 16, 32, 64], constants.DataWidth]*) – Number of bits to read per element.
- **extended** (*bool, optional*) – Use 64 bits offset independent of the platform, by default False.

Returns

- **data** (*List[int]*) – Data read from the bus

- **status_code** (*constants.StatusCode*) – Return value of the library call.

Raises ValueError – Raised if an invalid width is specified.

move_out(*space: pyvisa.constants.AddressSpace, offset: int, length: int, data: Iterable[int], width: pyvisa.constants.DataWidth, extended: bool = False*) → pyvisa.constants.StatusCode

Move a block of data from local memory to the given address space and offset.

Corresponds to viMoveOut* functions of the VISA library.

Parameters

- **space** (*constants.AddressSpace*) – Address space into which move the data.
- **offset** (*int*) – Offset (in bytes) of the address or register from which to read.
- **length** (*int*) – Number of elements to transfer, where the data width of the elements to transfer is identical to the source data width.
- **data** (*Iterable[int]*) – Data to write to bus.
- **width** (*Union[Literal[8, 16, 32, 64], constants.DataWidth]*) – Number of bits to per element.
- **extended** (*bool, optional*) – Use 64 bits offset independent of the platform, by default False.

Returns Return value of the library call.

Return type *constants.StatusCode*

Raises ValueError – Raised if an invalid width is specified.

open(*access_mode: pyvisa.constants.AccessModes = <AccessModes.no_lock: 0>, open_timeout: int = 5000*) → None

Opens a session to the specified resource.

Parameters

- **access_mode** (constants.AccessModes, *optional*) – Specifies the mode by which the resource is to be accessed. Defaults to constants.AccessModes.no_lock.
- **open_timeout** (*int, optional*) – If the access_mode parameter requests a lock, then this parameter specifies the absolute time period (in milliseconds) that the resource waits to get unlocked before this operation returns an error. Defaults to 5000.

read_memory(*space: pyvisa.constants.AddressSpace, offset: int, width: pyvisa.constants.DataWidth, extended: bool = False*) → int

Read a value from the specified memory space and offset.

Parameters

- **space** (*constants.AddressSpace*) – Specifies the address space from which to read.
- **offset** (*int*) – Offset (in bytes) of the address or register from which to read.
- **width** (*Union[Literal[8, 16, 32, 64], constants.DataWidth]*) – Number of bits to read (8, 16, 32 or 64).
- **extended** (*bool, optional*) – Use 64 bits offset independent of the platform.

Returns **data** – Data read from memory

Return type int

Raises ValueError – Raised if an invalid width is specified.

classmethod register(*interface_type: pyvisa.constants.InterfaceType*, *resource_class: str*) → Callable[[Type[T]], Type[T]]

Create a decorator to register a class.

The class is associated to an interface type, resource class pair.

Parameters

- **interface_type** (constants.InterfaceType) – Interface type for which to register a wrapper class.
- **resource_class** (*str*) – Resource class for which to register a wrapper class.

Returns Decorator registering the class. Raises TypeError if some VISA attributes are missing on the registered class.

Return type Callable[[Type[T]], Type[T]]

resource_class

Resource class as defined by the canonical resource name.

Possible values are: INSTR, INTFC, SOCKET, RAW...

VISA Attribute VI_ATTR_RSRC_CLASS (3221159937)

resource_info

Get the extended information of this resource.

resource_manufacturer_name

Manufacturer name of the vendor that implemented the VISA library.

This attribute is not related to the device manufacturer attributes.

Note The value of this attribute is for display purposes only and not for programmatic decisions, as the value can differ between VISA implementations and/or revisions.

VISA Attribute VI_ATTR_RSRC_MANF_NAME (3221160308)

resource_name

Unique identifier for a resource compliant with the address structure. :VISA Attribute: VI_ATTR_RSRC_NAME (3221159938)

session

Resource session handle.

Raises errors.InvalidSession – Raised if session is closed.

set_visa_attribute(*name: pyvisa.constants.ResourceAttribute*, *state: Any*) → pyvisa.constants.StatusCode

Set the state of an attribute.

One should prefer the dedicated descriptor for often used attributes since those perform checks and automatic conversion on the value.

Parameters

- **name** (*constants.ResourceAttribute*) – Attribute for which the state is to be modified.
- **state** (*Any*) – The state of the attribute to be set for the specified object.

Returns Return value of the library call.

Return type *constants.StatusCode*

spec_version
Version of the VISA specification to which the implementation is compliant.

The format of the value has the upper 12 bits as the major number of the version, the next lower 12 bits as the minor number of the version, and the lowest 8 bits as the sub-minor number of the version. The current VISA specification defines the value to be 00300000h.

VISA Attribute VI_ATTR_RSRC_SPEC_VERSION (1073676656)

Type int

Range 0 <= value <= 4294967295

timeout
Timeout in milliseconds for all resource I/O operations.

This value is used when accessing the device associated with the given session.

Special values:

- **immediate (`VI_TMO_IMMEDIATE`): 0** (for convenience, any value smaller than 1 is considered as 0)
- **infinite (`VI_TMO_INFINITE`): `float('+inf')`** (for convenience, None is considered as `float('+inf')`)

To set an **infinite** timeout, you can also use:

```
>>> del instrument.timeout
```

A timeout value of VI_TMO_IMMEDIATE means that operations should never wait for the device to respond. A timeout value of VI_TMO_INFINITE disables the timeout mechanism.

VISA Attribute VI_ATTR_TMO_VALUE (1073676314)

Type int

Range 0 <= value <= 4294967295

uninstall_handler(*event_type: pyvisa.constants.EventType, handler: Callable[[NewType.<locals>.new_type, pyvisa.constants.EventType, NewType.<locals>.new_type, Any], None], user_handle=None*) → None
Uninstalls handlers for events in this resource.

Parameters

- **event_type** (*constants.EventType*) – Logical event identifier.
- **handler** (*VISAHandler*) – Handler function to be uninstalled by a client application.
- **user_handle** (*Any*) – The user handle returned by install_handler.

unlock() → None
Relinquishes a lock for the specified resource.

visa_attributes_classes = {<class 'pyvisa.attributes.AttrVI_ATTR_RSRC_NAME'>, <class '

wait_on_event(*in_event_type: pyvisa.constants.EventType, timeout: int, capture_timeout: bool = False*) → pyvisa.resources.resource.WaitResponse
Waits for an occurrence of the specified event in this resource.

in_event_type [constants.EventType] Logical identifier of the event(s) to wait for.

timeout [int] Absolute time period in time units that the resource shall wait for a specified event to occur before returning the time elapsed error. The time unit is in milliseconds. None means waiting forever if necessary.

capture_timeout [bool, optional] When True will not produce a VisaIOError(VI_ERROR_TMO) but instead return a WaitResponse with timed_out=True.

Returns Object that contains event_type, context and ret value.

Return type WaitResponse

wrap_handler(*callable: Callable[[Resource, pyvisa.events.Event, Any], None]*) → Callable[[NewType.<locals>.new_type, pyvisa.constants.EventType, NewType.<locals>.new_type, Any], None]

Wrap an event handler to provide the signature expected by VISA.

The handler is expected to have the following signature: handler(resource: Resource, event: Event, user_handle: Any) -> None.

The wrapped handler should be used only to handle events on the resource used to wrap the handler.

write_memory(*space: pyvisa.constants.AddressSpace*, *offset: int*, *data: int*, *width: pyvisa.constants.DataWidth*, *extended: bool = False*) → pyvisa.constants.StatusCode

Write a value to the specified memory space and offset.

Parameters

- **space** (*constants.AddressSpace*) – Specifies the address space.
- **offset** (*int*) – Offset (in bytes) of the address or register from which to read.
- **data** (*int*) – Data to write to bus.
- **width** (*Union[Literal[8, 16, 32, 64], constants.DataWidth]*) – Number of bits to read.
- **extended** (*bool, optional*) – Use 64 bits offset independent of the platform, by default False.

Returns Return value of the library call.

Return type *constants.StatusCode*

Raises ValueError – Raised if an invalid width is specified.

class pyvisa.resources.**SerialInstrument**(*resource_manager: pyvisa.highlevel.ResourceManager*, *resource_name: str*)

Communicates with devices of type ASRL<board>[::INSTR]

Do not instantiate directly, use *pyvisa.highlevel.ResourceManager.open_resource()*.

CR = '\r'

LF = '\n'

allow_dma

Should I/O accesses use DMA (True) or Programmed I/O (False).

In some implementations, this attribute may have global effects even though it is documented to be a local attribute. Since this affects performance and not functionality, that behavior is acceptable.

VISA Attribute VI_ATTR_DMA_ALLOW_EN (1073676318)

Type bool

allow_transmit
Manually control transmission.

assert_trigger() → None
Sends a software trigger to the device.

baud_rate
Baud rate of the interface.

before_close() → None
Called just before closing an instrument.

break_length
Duration (in milliseconds) of the break signal.

break_state
Manually control the assertion state of the break signal.

bytes_in_buffer
Number of bytes available in the low- level I/O receive buffer.

chunk_size = 20480

clear() → None
Clear this resource.

close() → None
Closes the VISA session and marks the handle as invalid.

data_bits
Number of data bits contained in each frame (from 5 to 8).

disable_event(*event_type: pyvisa.constants.EventType*, *mechanism: pyvisa.constants.EventMechanism*) → None
Disable notification for an event type(s) via the specified mechanism(s).

Parameters

- **event_type** (*constants.EventType*) – Logical event identifier.
- **mechanism** (*constants.EventMechanism*) – Specifies event handling mechanisms to be disabled.

discard_events(*event_type: pyvisa.constants.EventType*, *mechanism: pyvisa.constants.EventMechanism*) → None
Discards event occurrences for an event type and mechanism in this resource.

Parameters

- **event_type** (*constants.EventType*) – Logical event identifier.
- **mechanism** (*constants.EventMechanism*) – Specifies event handling mechanisms to be disabled.

discard_null
If set to True, NUL characters are discarded.

enable_event(*event_type: pyvisa.constants.EventType*, *mechanism: pyvisa.constants.EventMechanism*, *context: None = None*) → None
Enable event occurrences for specified event types and mechanisms in this resource.

Parameters

- **event_type** (*constants.EventType*) – Logical event identifier.
- **mechanism** (*constants.EventMechanism*) – Specifies event handling mechanisms to be enabled
- **context** (*None*) – Not currently used, leave as None.

encoding
Encoding used for read and write operations.

end_input
Method used to terminate read operations.

end_output
Method used to terminate write operations.

flow_control
Indicates the type of flow control used by the transfer mechanism.

flush(*mask: pyvisa.constants.BufferOperation*) → None
Manually clears the specified buffers.

Depending on the value of the mask this can cause the buffer data to be written to the device.

Parameters **mask** (*constants.BufferOperation*) – Specifies the action to be taken with flushing the buffer. See highlevel.VisaLibraryBase.flush for a detailed description.

get_visa_attribute(*name: pyvisa.constants.ResourceAttribute*) → Any
Retrieves the state of an attribute in this resource.

One should prefer the dedicated descriptor for often used attributes since those perform checks and automatic conversion on the value.

Parameters **name** (*constants.ResourceAttribute*) – Resource attribute for which the state query is made.

Returns The state of the queried attribute for a specified resource.

Return type Any

ignore_warning(**warnings_constants*) → AbstractContextManager[T_co]
Ignoring warnings context manager for the current resource.

Parameters **warnings_constants** (constants.StatusCode) – Constants identifying the warnings to ignore.

implementation_version
Resource version that identifies the revisions or implementations of a resource.

This attribute value is defined by the individual manufacturer and increments with each new revision. The format of the value has the upper 12 bits as the major number of the version, the next lower 12 bits as the minor number of the version, and the lowest 8 bits as the sub-minor number of the version.

VISA Attribute VI_ATTR_RSRC_IMPL_VERSION (1073676291)

Type int

Range 0 <= value <= 4294967295

install_handler(*event_type: pyvisa.constants.EventType, handler: Callable[[NewType.<locals>.new_type, pyvisa.constants.EventType, NewType.<locals>.new_type, Any], None], user_handle=None*) → Any
Install handlers for event callbacks in this resource.

Parameters

- **event_type** (*constants.EventType*) – Logical event identifier.
- **handler** (*VISAHandler*) – Handler function to be installed by a client application.
- **user_handle** – A value specified by an application that can be used for identifying handlers uniquely for an event type. Depending on the backend they may be restriction on the possible values. Look at the backend *install_visa_handler* for more details.

Returns User handle in a format amenable to the backend. This is this representation of the handle that should be used when unistalling a handler.

Return type Any

interface_number
Board number for the given interface. :VISA Attribute: VI_ATTR_INTF_NUM (1073676662) :type: int :range: 0 <= value <= 65535

interface_type
Interface type of the given session. :VISA Attribute: VI_ATTR_INTF_TYPE (1073676657) :type: :class:pyvisa.constants.InterfaceType

io_protocol
IO protocol to use.

In VXI, you can choose normal word serial or fast data channel (FDC). In GPIB, you can choose normal or high-speed (HS-488) transfers. In serial, TCPIP, or USB RAW, you can choose normal transfers or 488.2-defined strings. In USB INSTR, you can choose normal or vendor-specific transfers.

VISA Attribute VI_ATTR_IO_PROT (1073676316)

Type :class:pyvisa.constants.IOProtocol

last_status
Last status code for this session.

lock(*timeout: Union[float, typing_extensions.Literal['default']][default]] = 'default', requested_key: Optional[str] = None*) → str
Establish a shared lock to the resource.

Parameters

- **timeout** (*Union[float, Literal["default"]], optional*) – Absolute time period (in milliseconds) that a resource waits to get unlocked by the locking session before returning an error. Defaults to "default" which means use self.timeout.
- **requested_key** (*Optional[str], optional*) – Access key used by another session with which you want your session to share a lock or None to generate a new shared access key.

Returns A new shared access key if requested_key is None, otherwise, same value as the requested_key

Return type str

lock_context(*timeout: Union[float, typing_extensions.Literal['default']][default]] = 'default', requested_key: Optional[str] = 'exclusive'*) → Iterator[Optional[str]]
A context that locks

Parameters

- **timeout** (*Union[float, Literal["default"]], optional*) – Absolute time period (in milliseconds) that a resource waits to get unlocked by the locking session before returning an error. Defaults to "default" which means use self.timeout.
- **requested_key** (*Optional[str], optional*) – When using default of 'exclusive' the lock is an exclusive lock. Otherwise it is the access key for the shared lock or None to generate a new shared access key.

Yields *Optional[str]* – The access_key if applicable.

lock_excl(*timeout: Union[float, typing_extensions.Literal['default']][default]] = 'default'*) → None
Establish an exclusive lock to the resource.

Parameters **timeout** (*Union[float, Literal["default"]], optional*) – Absolute time period (in milliseconds) that a resource waits to get unlocked by the locking session before returning an error. Defaults to "default" which means use self.timeout.

lock_state
Current locking state of the resource.

The resource can be unlocked, locked with an exclusive lock, or locked with a shared lock.

VISA Attribute VI_ATTR_RSRC_LOCK_STATE (1073676292)

Type :class:pyvisa.constants.AccessModes

open(*access_mode: pyvisa.constants.AccessModes = <AccessModes.no_lock: 0>, open_timeout: int = 5000*) → None
Opens a session to the specified resource.

Parameters

- **access_mode** (*constants.AccessModes, optional*) – Specifies the mode by which the resource is to be accessed. Defaults to constants.AccessModes.no_lock.
- **open_timeout** (*int, optional*) – If the access_mode parameter requests a lock, then this parameter specifies the absolute time period (in milliseconds) that the resource waits to get unlocked before this operation returns an error. Defaults to 5000.

parity
Parity used with every frame transmitted and received.

query(*message: str, delay: Optional[float] = None*) → str
A combination of write(message) and read()

Parameters

- **message** (*str*) – The message to send.
- **delay** (*Optional[float], optional*) – Delay in seconds between write and read operations. If None, defaults to self.query_delay.

Returns Answer from the device.

Return type str

query_ascii_values(*message: str, converter: Union[typing_extensions.Literal['s', 'b', 'c', 'd', 'o', 'x', 'X', 'e', 'E', 'f', 'F', 'g', 'G'][s, b, c, d, o, x, X, e, E, f, F, g, G], Callable[[str], Any]] = 'f', separator: Union[str, Callable[[str], Iterable[str]]] = ',', container: Union[Type[CT_co], Callable[[Iterable[T_co]], Sequence[T_co]]] = <class 'list'>, delay: Optional[float] = None*) → Sequence[Any]
Query the device for values in ascii format returning an iterable of values.

Parameters

- **message** (*str*) – The message to send.
- **converter** (*ASCII_CONVERTER, optional*) – Str format of function to convert each value. Default to "f".
- **separator** (*Union[str, Callable[[str], Iterable[str]]]*) – str or callable used to split the data into individual elements. If a str is given, data.split(separator) is used. Default to ",".
- **container** (*Union[Type, Callable[[Iterable], Sequence]], optional*) – Container type to use for the output data. Possible values are: list, tuple, np.ndarray, etc, Default to list.
- **delay** (*Optional[float], optional*) – Delay in seconds between write and read operations. If None, defaults to self.query_delay.

Returns Parsed data.

Return type Sequence

query_binary_values(*message: str, datatype: typing_extensions.Literal['s', 'b', 'B', 'h', 'H', 'i', 'I', 'l', 'L', 'q', 'Q', 'f', 'd'][s, b, B, h, H, i, I, l, L, q, Q, f, d] = 'f', is_big_endian: bool = False, container: Union[Type[CT_co], Callable[[Iterable[T_co]], Sequence[T_co]]] = <class 'list'>, delay: Optional[float] = None, header_fmt: typing_extensions.Literal['ieee', 'hp', 'empty'][ieee, hp, empty] = 'ieee', expect_termination: bool = True, data_points: int = 0, chunk_size: Optional[int] = None*) → Sequence[Union[int, float]]

Query the device for values in binary format returning an iterable of values.

Parameters

- **message** (*str*) – The message to send.
- **datatype** (*BINARY_DATATYPES, optional*) – Format string for a single element. See struct module. 'f' by default.
- **is_big_endian** (*bool, optional*) – Are the data in big or little endian order. Defaults to False.
- **container** (*Union[Type, Callable[[Iterable], Sequence]], optional*) – Container type to use for the output data. Possible values are: list, tuple, np.ndarray, etc, Default to list.
- **delay** (*Optional[float], optional*) – Delay in seconds between write and read operations. If None, defaults to self.query_delay.
- **header_fmt** (*util.BINARY_HEADERS, optional*) – Format of the header prefixing the data. Defaults to 'ieee'.
- **expect_termination** (*bool, optional*) – When set to False, the expected length of the binary values block does not account for the final termination character (the read termination). Defaults to True.
- **data_points** (*int, optional*) – Number of points expected in the block. This is used only if the instrument does not report it itself. This will be converted in a number of bytes based on the datatype. Defaults to 0.
- **chunk_size** (*int, optional*) – Size of the chunks to read from the device. Using larger chunks may be faster for large amount of data.

Returns Data read from the device.

Return type Sequence[Union[int, float]]

query_delay = 0.0

read(*termination: Optional[str] = None*, *encoding: Optional[str] = None*) → str
Read a string from the device.

Reading stops when the device stops sending (e.g. by setting appropriate bus lines), or the termination characters sequence was detected. Attention: Only the last character of the termination characters is really used to stop reading, however, the whole sequence is compared to the ending of the read string message. If they don't match, a warning is issued.

Parameters

- **termination** (*Optional[str], optional*) – Alternative character termination to use. If None, the value of write_termination is used. Defaults to None.
- **encoding** (*Optional[str], optional*) – Alternative encoding to use to turn bytes into str. If None, the value of encoding is used. Defaults to None.

Returns Message read from the instrument and decoded.

Return type str

read_ascii_values(*converter: Union[typing_extensions.Literal['s', 'b', 'c', 'd', 'o', 'x', 'X', 'e', 'E', 'f', 'F', 'g', 'G'][s, b, c, d, o, x, X, e, E, f, F, g, G], Callable[[str], Any]] = 'f', separator: Union[str, Callable[[str], Iterable[str]]] = ',', container: Union[Type[CT_co], Callable[[Iterable[T_co]], Sequence[T_co]]] = <class 'list'>*) → Sequence[T_co]
Read values from the device in ascii format returning an iterable of values.

Parameters **converter** (*ASCII_CONVERTER, optional*) – Str format of function to convert each value. Default to "f".

separator [Union[str, Callable[[str], Iterable[str]]]] str or callable used to split the data into individual elements. If a str is given, data.split(separator) is used. Default to ",".

container [Union[Type, Callable[[Iterable], Sequence]], optional] Container type to use for the output data. Possible values are: list, tuple, np.ndarray, etc, Default to list.

Returns Parsed data.

Return type Sequence

read_binary_values(*datatype: typing_extensions.Literal['s', 'b', 'B', 'h', 'H', 'i', 'I', 'l', 'L', 'q', 'Q', 'f', 'd'][s, b, B, h, H, i, I, l, L, q, Q, f, d] = 'f', is_big_endian: bool = False, container: Union[Type[CT_co], Callable[[Iterable[T_co]], Sequence[T_co]]] = <class 'list'>, header_fmt: typing_extensions.Literal['ieee', 'hp', 'empty'][ieee, hp, empty] = 'ieee', expect_termination: bool = True, data_points: int = 0, chunk_size: Optional[int] = None*) → Sequence[Union[int, float]]
Read values from the device in binary format returning an iterable of values.

Parameters

- **datatype** (*BINARY_DATATYPES, optional*) – Format string for a single element. See struct module. 'f' by default.
- **is_big_endian** (*bool, optional*) – Are the data in big or little endian order. Defaults to False.

- **container** (*Union[Type, Callable[[Iterable], Sequence]], optional*) – Container type to use for the output data. Possible values are: list, tuple, np.ndarray, etc, Default to list.
- **header_fmt** (*util.BINARY_HEADERS, optional*) – Format of the header prefixing the data. Defaults to 'ieee'.
- **expect_termination** (*bool, optional*) – When set to False, the expected length of the binary values block does not account for the final termination character (the read termination). Defaults to True.
- **data_points** (*int, optional*) – Number of points expected in the block. This is used only if the instrument does not report it itself. This will be converted in a number of bytes based on the datatype. Defaults to 0.
- **chunk_size** (*int, optional*) – Size of the chunks to read from the device. Using larger chunks may be faster for large amount of data.

Returns Data read from the device.

Return type Sequence[Union[int, float]]

read_bytes(*count: int*, *chunk_size: Optional[int] = None*, *break_on_termchar: bool = False*) → bytes

Read a certain number of bytes from the instrument.

Parameters

- **count** (*int*) – The number of bytes to read from the instrument.
- **chunk_size** (*Optional[int], optional*) – The chunk size to use to perform the reading. If count > chunk_size multiple low level operations will be performed. Defaults to None, meaning the resource wide set value is set.
- **break_on_termchar** (*bool, optional*) – Should the reading stop when a termination character is encountered or when the message ends. Defaults to False.

Returns Bytes read from the instrument.

Return type bytes

read_raw(*size: Optional[int] = None*) → bytes

Read the unmodified string sent from the instrument to the computer.

In contrast to read(), no termination characters are stripped.

Parameters **size** (*Optional[int], optional*) – The chunk size to use to perform the reading. Defaults to None, meaning the resource wide set value is set.

Returns Bytes read from the instrument.

Return type bytes

read_stb() → int

Service request status register.

read_termination

Read termination character.

read_termination_context(*new_termination: str*) → Iterator[T_co]

classmethod register(*interface_type: pyvisa.constants.InterfaceType*, *resource_class: str*) → Callable[[Type[T]], Type[T]]

Create a decorator to register a class.

The class is associated to an interface type, resource class pair.

Parameters

- **interface_type** (constants.InterfaceType) – Interface type for which to register a wrapper class.
- **resource_class** (*str*) – Resource class for which to register a wrapper class.

Returns Decorator registering the class. Raises TypeError if some VISA attributes are missing on the registered class.

Return type Callable[[Type[T]], Type[T]]

replace_char
Character to be used to replace incoming characters that arrive with errors.

resource_class
Resource class as defined by the canonical resource name.

Possible values are: INSTR, INTFC, SOCKET, RAW...

VISA Attribute VI_ATTR_RSRC_CLASS (3221159937)

resource_info
Get the extended information of this resource.

resource_manufacturer_name
Manufacturer name of the vendor that implemented the VISA library.

This attribute is not related to the device manufacturer attributes.

Note The value of this attribute is for display purposes only and not for programmatic decisions, as the value can differ between VISA implementations and/or revisions.

VISA Attribute VI_ATTR_RSRC_MANF_NAME (3221160308)

resource_name
Unique identifier for a resource compliant with the address structure. :VISA Attribute: VI_ATTR_RSRC_NAME (3221159938)

send_end
Should END be asserted during the transfer of the last byte of the buffer. :VISA Attribute: VI_ATTR_SEND_END_EN (1073676310) :type: bool

session
Resource session handle.

Raises errors.InvalidSession – Raised if session is closed.

set_visa_attribute(*name: pyvisa.constants.ResourceAttribute*, *state: Any*) → pyvisa.constants.StatusCode
Set the state of an attribute.

One should prefer the dedicated descriptor for often used attributes since those perform checks and automatic conversion on the value.

Parameters

- **name** (*constants.ResourceAttribute*) – Attribute for which the state is to be modified.
- **state** (*Any*) – The state of the attribute to be set for the specified object.

Returns Return value of the library call.

Return type *constants.StatusCode*

spec_version
Version of the VISA specification to which the implementation is compliant.

The format of the value has the upper 12 bits as the major number of the version, the next lower 12 bits as the minor number of the version, and the lowest 8 bits as the sub-minor number of the version. The current VISA specification defines the value to be 00300000h.

VISA Attribute VI_ATTR_RSRC_SPEC_VERSION (1073676656)

Type int

Range 0 <= value <= 4294967295

stb
Service request status register.

stop_bits
Number of stop bits used to indicate the end of a frame.

timeout
Timeout in milliseconds for all resource I/O operations.

This value is used when accessing the device associated with the given session.

Special values:

- **immediate (VI_TMO_IMMEDIATE): 0** (for convenience, any value smaller than 1 is considered as 0)
- **infinite (VI_TMO_INFINITE): float('+inf')** (for convenience, None is considered as `float('+inf')`)

To set an **infinite** timeout, you can also use:

```
>>> del instrument.timeout
```

A timeout value of VI_TMO_IMMEDIATE means that operations should never wait for the device to respond. A timeout value of VI_TMO_INFINITE disables the timeout mechanism.

VISA Attribute VI_ATTR_TMO_VALUE (1073676314)

Type int

Range 0 <= value <= 4294967295

uninstall_handler(*event_type: pyvisa.constants.EventType, handler: Callable[[NewType.<locals>.new_type, pyvisa.constants.EventType, NewType.<locals>.new_type, Any], None], user_handle=None*) → None
Uninstalls handlers for events in this resource.

Parameters

- **event_type** (*constants.EventType*) – Logical event identifier.
- **handler** (*VISAHandler*) – Handler function to be uninstalled by a client application.
- **user_handle** (*Any*) – The user handle returned by install_handler.

unlock() → None
Relinquishes a lock for the specified resource.

visa_attributes_classes = {<class 'pyvisa.attributes.AttrVI_ATTR_INTF_INST_NAME'>, <cl

wait_on_event(*in_event_type: pyvisa.constants.EventType*, *timeout: int*, *capture_timeout: bool = False*) → pyvisa.resources.resource.WaitResponse

Waits for an occurrence of the specified event in this resource.

in_event_type [constants.EventType] Logical identifier of the event(s) to wait for.

timeout [int] Absolute time period in time units that the resource shall wait for a specified event to occur before returning the time elapsed error. The time unit is in milliseconds. None means waiting forever if necessary.

capture_timeout [bool, optional] When True will not produce a VisaIOError(VI_ERROR_TMO) but instead return a WaitResponse with timed_out=True.

Returns Object that contains event_type, context and ret value.

Return type WaitResponse

wrap_handler(*callable: Callable[[Resource, pyvisa.events.Event, Any], None]*) → Callable[[NewType.<locals>.new_type, pyvisa.constants.EventType, NewType.<locals>.new_type, Any], None]

Wrap an event handler to provide the signature expected by VISA.

The handler is expected to have the following signature: handler(resource: Resource, event: Event, user_handle: Any) -> None.

The wrapped handler should be used only to handle events on the resource used to wrap the handler.

write(*message: str*, *termination: Optional[str] = None*, *encoding: Optional[str] = None*) → int

Write a string message to the device.

The write_termination is always appended to it.

Parameters

- **message** (*str*) – The message to be sent.
- **termination** (*Optional[str], optional*) – Alternative character termination to use. If None, the value of write_termination is used. Defaults to None.
- **encoding** (*Optional[str], optional*) – Alternative encoding to use to turn str into bytes. If None, the value of encoding is used. Defaults to None.

Returns Number of bytes written.

Return type int

write_ascii_values(*message: str, values: Sequence[Any], converter: Union[typing_extensions.Literal['s', 'b', 'c', 'd', 'o', 'x', 'X', 'e', 'E', 'f', 'F', 'g', 'G'][s, b, c, d, o, x, X, e, E, f, F, g, G], Callable[[str], Any]] = 'f', separator: Union[str, Callable[[Iterable[str]], str]] = ',', termination: Optional[str] = None, encoding: Optional[str] = None*)

Write a string message to the device followed by values in ascii format.

The write_termination is always appended to it.

Parameters

- **message** (*str*) – Header of the message to be sent.
- **values** (*Sequence[Any]*) – Data to be writen to the device.
- **converter** (*Union[str, Callable[[Any], str]], optional*) – Str formatting codes or function used to convert each value. Defaults to "f".

- **separator** (*Union[str, Callable[[Iterable[str]], str]], optional*) – Str or callable that join the values in a single str. If a str is given, separator.join(values) is used. Defaults to ‘,’
- **termination** (*Optional[str], optional*) – Alternative character termination to use. If None, the value of write_termination is used. Defaults to None.
- **encoding** (*Optional[str], optional*) – Alternative encoding to use to turn str into bytes. If None, the value of encoding is used. Defaults to None.

Returns Number of bytes written.

Return type int

write_binary_values(*message: str, values: Sequence[Any], datatype: typing_extensions.Literal['s', 'b', 'B', 'h', 'H', 'i', 'I', 'l', 'L', 'q', 'Q', 'f', 'd'][s, b, B, h, H, i, I, l, L, q, Q, f, d] = 'f', is_big_endian: bool = False, termination: Optional[str] = None, encoding: Optional[str] = None, header_fmt: typing_extensions.Literal['ieee', 'hp', 'empty'][ieee, hp, empty] = 'ieee'*)

Write a string message to the device followed by values in binary format.

The write_termination is always appended to it.

Parameters

- **message** (*str*) – The header of the message to be sent.
- **values** (*Sequence[Any]*) – Data to be written to the device.
- **datatype** (*util.BINARY_DATATYPES, optional*) – The format string for a single element. See struct module.
- **is_big_endian** (*bool, optional*) – Are the data in big or little endian order.
- **termination** (*Optional[str], optional*) – Alternative character termination to use. If None, the value of write_termination is used. Defaults to None.
- **encoding** (*Optional[str], optional*) – Alternative encoding to use to turn str into bytes. If None, the value of encoding is used. Defaults to None.
- **header_fmt** (*util.BINARY_HEADERS*) – Format of the header prefixing the data.

Returns Number of bytes written.

Return type int

write_raw(*message: bytes*) → int

Write a byte message to the device.

Parameters **message** (*bytes*) – The message to be sent.

Returns Number of bytes written

Return type int

write_termination

Write termination character.

xoff_char

XOFF character used for XON/XOFF flow control (both directions).

xon_char

XON character used for XON/XOFF flow control (both directions).

class pyvisa.resources.**TCPIPInstrument**(*resource_manager: pyvisa.highlevel.ResourceManager*, *resource_name: str*)

Communicates with to devices of type TCPIP::host address[::INSTR]

More complex resource names can be specified with the following grammar: TCPIP[board]::host address[::LAN device name][::INSTR]

Do not instantiate directly, use *pyvisa.highlevel.ResourceManager.open_resource()*.

CR = '\r'

LF = '\n'

allow_dma

Should I/O accesses use DMA (True) or Programmed I/O (False).

In some implementations, this attribute may have global effects even though it is documented to be a local attribute. Since this affects performance and not functionality, that behavior is acceptable.

VISA Attribute VI_ATTR_DMA_ALLOW_EN (1073676318)

Type bool

assert_trigger() → None

Sends a software trigger to the device.

before_close() → None

Called just before closing an instrument.

chunk_size = 20480

clear() → None

Clear this resource.

close() → None

Closes the VISA session and marks the handle as invalid.

control_ren(*mode: pyvisa.constants.RENLineOperation*) → pyvisa.constants.StatusCode

Controls the state of the GPIB Remote Enable (REN) interface line.

The remote/local state of the device can also be controlled optionally.

Corresponds to viGpibControlREN function of the VISA library.

Parameters **mode** (*constants.RENLineOperation*) – Specifies the state of the REN line and optionally the device remote/local state.

Returns Return value of the library call.

Return type *constants.StatusCode*

disable_event(*event_type: pyvisa.constants.EventType*, *mechanism: pyvisa.constants.EventMechanism*) → None

Disable notification for an event type(s) via the specified mechanism(s).

Parameters

- **event_type** (*constants.EventType*) – Logical event identifier.
- **mechanism** (*constants.EventMechanism*) – Specifies event handling mechanisms to be disabled.

discard_events(*event_type: pyvisa.constants.EventType*, *mechanism: pyvisa.constants.EventMechanism*) → None
Discards event occurrences for an event type and mechanism in this resource.

Parameters

- **event_type** (*constants.EventType*) – Logical event identifier.
- **mechanism** (*constants.EventMechanism*) – Specifies event handling mechanisms to be disabled.

enable_event(*event_type: pyvisa.constants.EventType*, *mechanism: pyvisa.constants.EventMechanism*, *context: None = None*) → None
Enable event occurrences for specified event types and mechanisms in this resource.

Parameters

- **event_type** (*constants.EventType*) – Logical event identifier.
- **mechanism** (*constants.EventMechanism*) – Specifies event handling mechanisms to be enabled
- **context** (*None*) – Not currently used, leave as None.

encoding
Encoding used for read and write operations.

flush(*mask: pyvisa.constants.BufferOperation*) → None
Manually clears the specified buffers.

Depending on the value of the mask this can cause the buffer data to be written to the device.

Parameters **mask** (*constants.BufferOperation*) – Specifies the action to be taken with flushing the buffer. See highlevel.VisaLibraryBase.flush for a detailed description.

get_visa_attribute(*name: pyvisa.constants.ResourceAttribute*) → Any
Retrieves the state of an attribute in this resource.

One should prefer the dedicated descriptor for often used attributes since those perform checks and automatic conversion on the value.

Parameters **name** (*constants.ResourceAttribute*) – Resource attribute for which the state query is made.

Returns The state of the queried attribute for a specified resource.

Return type Any

ignore_warning(**warnings_constants*) → AbstractContextManager[T_co]
Ignoring warnings context manager for the current resource.

Parameters **warnings_constants** (constants.StatusCode) – Constants identifying the warnings to ignore.

implementation_version
Resource version that identifies the revisions or implementations of a resource.

This attribute value is defined by the individual manufacturer and increments with each new revision. The format of the value has the upper 12 bits as the major number of the version, the next lower 12 bits as the minor number of the version, and the lowest 8 bits as the sub-minor number of the version.

VISA Attribute VI_ATTR_RSRC_IMPL_VERSION (1073676291)

Type int

Range 0 <= value <= 4294967295

install_handler(*event_type: pyvisa.constants.EventType, handler: Callable[[NewType.<locals>.new_type, pyvisa.constants.EventType, NewType.<locals>.new_type, Any], None], user_handle=None*) → Any

Install handlers for event callbacks in this resource.

Parameters

- **event_type** (*constants.EventType*) – Logical event identifier.
- **handler** (*VISAHandler*) – Handler function to be installed by a client application.
- **user_handle** – A value specified by an application that can be used for identifying handlers uniquely for an event type. Depending on the backend they may be restriction on the possible values. Look at the backend *install_visa_handler* for more details.

Returns User handle in a format amenable to the backend. This is this representation of the handle that should be used when unistalling a handler.

Return type Any

interface_number

Board number for the given interface. :VISA Attribute: VI_ATTR_INTF_NUM (1073676662) :type: int :range: 0 <= value <= 65535

interface_type

Interface type of the given session. :VISA Attribute: VI_ATTR_INTF_TYPE (1073676657) :type: :class:pyvisa.constants.InterfaceType

io_protocol

IO protocol to use.

In VXI, you can choose normal word serial or fast data channel (FDC). In GPIB, you can choose normal or high-speed (HS-488) transfers. In serial, TCPIP, or USB RAW, you can choose normal transfers or 488.2-defined strings. In USB INSTR, you can choose normal or vendor-specific transfers.

VISA Attribute VI_ATTR_IO_PROT (1073676316)

Type :class:pyvisa.constants.IOProtocol

last_status

Last status code for this session.

lock(*timeout: Union[float, typing_extensions.Literal['default'][default]] = 'default', requested_key: Optional[str] = None*) → str

Establish a shared lock to the resource.

Parameters

- **timeout** (*Union[float, Literal["default"]], optional*) – Absolute time period (in milliseconds) that a resource waits to get unlocked by the locking session before returning an error. Defaults to "default" which means use self.timeout.
- **requested_key** (*Optional[str], optional*) – Access key used by another session with which you want your session to share a lock or None to generate a new shared access key.

Returns A new shared access key if requested_key is None, otherwise, same value as the requested_key

Return type str

lock_context(*timeout: Union[float, typing_extensions.Literal['default'][default]] = 'default', requested_key: Optional[str] = 'exclusive'*) → Iterator[Optional[str]]
A context that locks

Parameters

- **timeout** (*Union[float, Literal["default"]], optional*) – Absolute time period (in milliseconds) that a resource waits to get unlocked by the locking session before returning an error. Defaults to "default" which means use self.timeout.
- **requested_key** (*Optional[str], optional*) – When using default of 'exclusive' the lock is an exclusive lock. Otherwise it is the access key for the shared lock or None to generate a new shared access key.

Yields *Optional[str]* – The access_key if applicable.

lock_excl(*timeout: Union[float, typing_extensions.Literal['default'][default]] = 'default'*) → None
Establish an exclusive lock to the resource.

Parameters **timeout** (*Union[float, Literal["default"]], optional*) – Absolute time period (in milliseconds) that a resource waits to get unlocked by the locking session before returning an error. Defaults to "default" which means use self.timeout.

lock_state
Current locking state of the resource.

The resource can be unlocked, locked with an exclusive lock, or locked with a shared lock.

VISA Attribute VI_ATTR_RSRC_LOCK_STATE (1073676292)

Type :class:pyvisa.constants.AccessModes

open(*access_mode: pyvisa.constants.AccessModes = <AccessModes.no_lock: 0>, open_timeout: int = 5000*) → None
Opens a session to the specified resource.

Parameters

- **access_mode** (*constants.AccessModes, optional*) – Specifies the mode by which the resource is to be accessed. Defaults to constants.AccessModes.no_lock.
- **open_timeout** (*int, optional*) – If the access_mode parameter requests a lock, then this parameter specifies the absolute time period (in milliseconds) that the resource waits to get unlocked before this operation returns an error. Defaults to 5000.

query(*message: str, delay: Optional[float] = None*) → str
A combination of write(message) and read()

Parameters

- **message** (*str*) – The message to send.
- **delay** (*Optional[float], optional*) – Delay in seconds between write and read operations. If None, defaults to self.query_delay.

Returns Answer from the device.

Return type str

query_ascii_values(*message: str, converter: Union[typing_extensions.Literal['s', 'b', 'c', 'd', 'o', 'x', 'X', 'e', 'E', 'f', 'F', 'g', 'G'][s, b, c, d, o, x, X, e, E, f, F, g, G], Callable[[str], Any]] = 'f', separator: Union[str, Callable[[str], Iterable[str]]] = ',', container: Union[Type[CT_co], Callable[[Iterable[T_co]], Sequence[T_co]]] = <class 'list'>, delay: Optional[float] = None*) → Sequence[Any]

Query the device for values in ascii format returning an iterable of values.

Parameters

- **message** (*str*) – The message to send.
- **converter** (*ASCII_CONVERTER, optional*) – Str format of function to convert each value. Default to "f".
- **separator** (*Union[str, Callable[[str], Iterable[str]]]*) – str or callable used to split the data into individual elements. If a str is given, data.split(separator) is used. Default to ",".
- **container** (*Union[Type, Callable[[Iterable], Sequence]], optional*) – Container type to use for the output data. Possible values are: list, tuple, np.ndarray, etc, Default to list.
- **delay** (*Optional[float], optional*) – Delay in seconds between write and read operations. If None, defaults to self.query_delay.

Returns Parsed data.

Return type Sequence

query_binary_values(*message: str, datatype: typing_extensions.Literal['s', 'b', 'B', 'h', 'H', 'i', 'I', 'l', 'L', 'q', 'Q', 'f', 'd'][s, b, B, h, H, i, I, l, L, q, Q, f, d] = 'f', is_big_endian: bool = False, container: Union[Type[CT_co], Callable[[Iterable[T_co]], Sequence[T_co]]] = <class 'list'>, delay: Optional[float] = None, header_fmt: typing_extensions.Literal['ieee', 'hp', 'empty'][ieee, hp, empty] = 'ieee', expect_termination: bool = True, data_points: int = 0, chunk_size: Optional[int] = None*) → Sequence[Union[int, float]]

Query the device for values in binary format returning an iterable of values.

Parameters

- **message** (*str*) – The message to send.
- **datatype** (*BINARY_DATATYPES, optional*) – Format string for a single element. See struct module. 'f' by default.
- **is_big_endian** (*bool, optional*) – Are the data in big or little endian order. Defaults to False.
- **container** (*Union[Type, Callable[[Iterable], Sequence]], optional*) – Container type to use for the output data. Possible values are: list, tuple, np.ndarray, etc, Default to list.
- **delay** (*Optional[float], optional*) – Delay in seconds between write and read operations. If None, defaults to self.query_delay.
- **header_fmt** (*util.BINARY_HEADERS, optional*) – Format of the header prefixing the data. Defaults to 'ieee'.
- **expect_termination** (*bool, optional*) – When set to False, the expected length of the binary values block does not account for the final termination character (the read termination). Defaults to True.

- **data_points** (*int, optional*) – Number of points expected in the block. This is used only if the instrument does not report it itself. This will be converted in a number of bytes based on the datatype. Defaults to 0.
- **chunk_size** (*int, optional*) – Size of the chunks to read from the device. Using larger chunks may be faster for large amount of data.

Returns Data read from the device.

Return type Sequence[Union[int, float]]

query_delay = 0.0

read(*termination: Optional[str] = None*, *encoding: Optional[str] = None*) → str

Read a string from the device.

Reading stops when the device stops sending (e.g. by setting appropriate bus lines), or the termination characters sequence was detected. Attention: Only the last character of the termination characters is really used to stop reading, however, the whole sequence is compared to the ending of the read string message. If they don't match, a warning is issued.

Parameters

- **termination** (*Optional[str], optional*) – Alternative character termination to use. If None, the value of write_termination is used. Defaults to None.
- **encoding** (*Optional[str], optional*) – Alternative encoding to use to turn bytes into str. If None, the value of encoding is used. Defaults to None.

Returns Message read from the instrument and decoded.

Return type str

read_ascii_values(*converter: Union[typing_extensions.Literal['s', 'b', 'c', 'd', 'o', 'x', 'X', 'e', 'E', 'f', 'F', 'g', 'G'][s, b, c, d, o, x, X, e, E, f, F, g, G], Callable[[str], Any]] = 'f', separator: Union[str, Callable[[str], Iterable[str]]] = ',', container: Union[Type[CT_co], Callable[[Iterable[T_co]], Sequence[T_co]]] = <class 'list'>*) → Sequence[T_co]

Read values from the device in ascii format returning an iterable of values.

Parameters **converter** (*ASCII_CONVERTER, optional*) – Str format of function to convert each value. Default to "f".

separator [Union[str, Callable[[str], Iterable[str]]]] str or callable used to split the data into individual elements. If a str is given, data.split(separator) is used. Default to ",".

container [Union[Type, Callable[[Iterable], Sequence]], optional] Container type to use for the output data. Possible values are: list, tuple, np.ndarray, etc, Default to list.

Returns Parsed data.

Return type Sequence

read_binary_values(*datatype: typing_extensions.Literal['s', 'b', 'B', 'h', 'H', 'i', 'I', 'l', 'L', 'q', 'Q', 'f', 'd'][s, b, B, h, H, i, I, l, L, q, Q, f, d] = 'f', is_big_endian: bool = False, container: Union[Type[CT_co], Callable[[Iterable[T_co]], Sequence[T_co]]] = <class 'list'>, header_fmt: typing_extensions.Literal['ieee', 'hp', 'empty'][ieee, hp, empty] = 'ieee', expect_termination: bool = True, data_points: int = 0, chunk_size: Optional[int] = None*) → Sequence[Union[int, float]]

Read values from the device in binary format returning an iterable of values.

Parameters

- **datatype** (*BINARY_DATATYPES, optional*) – Format string for a single element. See struct module. 'f' by default.
- **is_big_endian** (*bool, optional*) – Are the data in big or little endian order. Defaults to False.
- **container** (*Union[Type, Callable[[Iterable], Sequence]], optional*) – Container type to use for the output data. Possible values are: list, tuple, np.ndarray, etc, Default to list.
- **header_fmt** (*util.BINARY_HEADERS, optional*) – Format of the header prefixing the data. Defaults to 'ieee'.
- **expect_termination** (*bool, optional*) – When set to False, the expected length of the binary values block does not account for the final termination character (the read termination). Defaults to True.
- **data_points** (*int, optional*) – Number of points expected in the block. This is used only if the instrument does not report it itself. This will be converted in a number of bytes based on the datatype. Defaults to 0.
- **chunk_size** (*int, optional*) – Size of the chunks to read from the device. Using larger chunks may be faster for large amount of data.

Returns Data read from the device.

Return type Sequence[Union[int, float]]

read_bytes(*count: int*, *chunk_size: Optional[int] = None*, *break_on_termchar: bool = False*) → bytes

Read a certain number of bytes from the instrument.

Parameters

- **count** (*int*) – The number of bytes to read from the instrument.
- **chunk_size** (*Optional[int], optional*) – The chunk size to use to perform the reading. If count > chunk_size multiple low level operations will be performed. Defaults to None, meaning the resource wide set value is set.
- **break_on_termchar** (*bool, optional*) – Should the reading stop when a termination character is encountered or when the message ends. Defaults to False.

Returns Bytes read from the instrument.

Return type bytes

read_raw(*size: Optional[int] = None*) → bytes

Read the unmodified string sent from the instrument to the computer.

In contrast to read(), no termination characters are stripped.

Parameters **size** (*Optional[int], optional*) – The chunk size to use to perform the reading. Defaults to None, meaning the resource wide set value is set.

Returns Bytes read from the instrument.

Return type bytes

read_stb() → int

Service request status register.

read_termination

Read termination character.

read_termination_context(*new_termination: str*) → Iterator[T_co]

classmethod register(*interface_type: pyvisa.constants.InterfaceType*, *resource_class: str*) → Callable[[Type[T]], Type[T]]

Create a decorator to register a class.

The class is associated to an interface type, resource class pair.

Parameters

- **interface_type** (*constants.InterfaceType*) – Interface type for which to register a wrapper class.
- **resource_class** (*str*) – Resource class for which to register a wrapper class.

Returns Decorator registering the class. Raises TypeError if some VISA attributes are missing on the registered class.

Return type Callable[[Type[T]], Type[T]]

resource_class

Resource class as defined by the canonical resource name.

Possible values are: INSTR, INTFC, SOCKET, RAW...

VISA Attribute VI_ATTR_RSRC_CLASS (3221159937)

resource_info

Get the extended information of this resource.

resource_manufacturer_name

Manufacturer name of the vendor that implemented the VISA library.

This attribute is not related to the device manufacturer attributes.

Note The value of this attribute is for display purposes only and not for programmatic decisions, as the value can differ between VISA implementations and/or revisions.

VISA Attribute VI_ATTR_RSRC_MANF_NAME (3221160308)

resource_name

Unique identifier for a resource compliant with the address structure. :VISA Attribute: VI_ATTR_RSRC_NAME (3221159938)

send_end

Should END be asserted during the transfer of the last byte of the buffer. :VISA Attribute: VI_ATTR_SEND_END_EN (1073676310) :type: bool

session

Resource session handle.

Raises errors.InvalidSession – Raised if session is closed.

set_visa_attribute(*name: pyvisa.constants.ResourceAttribute*, *state: Any*) → pyvisa.constants.StatusCode

Set the state of an attribute.

One should prefer the dedicated descriptor for often used attributes since those perform checks and automatic conversion on the value.

Parameters

- **name** (*constants.ResourceAttribute*) – Attribute for which the state is to be modified.

- **state** (*Any*) – The state of the attribute to be set for the specified object.

Returns Return value of the library call.

Return type *constants.StatusCode*

spec_version
Version of the VISA specification to which the implementation is compliant.

The format of the value has the upper 12 bits as the major number of the version, the next lower 12 bits as the minor number of the version, and the lowest 8 bits as the sub-minor number of the version. The current VISA specification defines the value to be 00300000h.

VISA Attribute VI_ATTR_RSRC_SPEC_VERSION (1073676656)

Type int

Range 0 <= value <= 4294967295

stb
Service request status register.

timeout
Timeout in milliseconds for all resource I/O operations.

This value is used when accessing the device associated with the given session.

Special values:

- **immediate (VI_TMO_IMMEDIATE): 0** (for convenience, any value smaller than 1 is considered as 0)
- **infinite (VI_TMO_INFINITE): float('+inf')** (for convenience, None is considered as `float('+inf')`)

To set an **infinite** timeout, you can also use:

```
>>> del instrument.timeout
```

A timeout value of VI_TMO_IMMEDIATE means that operations should never wait for the device to respond. A timeout value of VI_TMO_INFINITE disables the timeout mechanism.

VISA Attribute VI_ATTR_TMO_VALUE (1073676314)

Type int

Range 0 <= value <= 4294967295

uninstall_handler(*event_type: pyvisa.constants.EventType, handler: Callable[[NewType.<locals>.new_type, pyvisa.constants.EventType, NewType.<locals>.new_type, Any], None], user_handle=None*) → None
Uninstalls handlers for events in this resource.

Parameters

- **event_type** (*constants.EventType*) – Logical event identifier.
- **handler** (*VISAHandler*) – Handler function to be uninstalled by a client application.
- **user_handle** (*Any*) – The user handle returned by install_handler.

unlock() → None
Relinquishes a lock for the specified resource.

visa_attributes_classes = {<class 'pyvisa.attributes.AttrVI_ATTR_TCPIP_IS_HISLIP'>, <c

wait_on_event(*in_event_type: pyvisa.constants.EventType*, *timeout: int*, *capture_timeout: bool = False*) → pyvisa.resources.resource.WaitResponse

Waits for an occurrence of the specified event in this resource.

in_event_type [constants.EventType] Logical identifier of the event(s) to wait for.

timeout [int] Absolute time period in time units that the resource shall wait for a specified event to occur before returning the time elapsed error. The time unit is in milliseconds. None means waiting forever if necessary.

capture_timeout [bool, optional] When True will not produce a VisaIOError(VI_ERROR_TMO) but instead return a WaitResponse with timed_out=True.

Returns Object that contains event_type, context and ret value.

Return type WaitResponse

wrap_handler(*callable: Callable[[Resource, pyvisa.events.Event, Any], None]*) → Callable[[NewType.<locals>.new_type, pyvisa.constants.EventType, NewType.<locals>.new_type, Any], None]

Wrap an event handler to provide the signature expected by VISA.

The handler is expected to have the following signature: handler(resource: Resource, event: Event, user_handle: Any) -> None.

The wrapped handler should be used only to handle events on the resource used to wrap the handler.

write(*message: str*, *termination: Optional[str] = None*, *encoding: Optional[str] = None*) → int

Write a string message to the device.

The write_termination is always appended to it.

Parameters

- **message** (*str*) – The message to be sent.
- **termination** (*Optional[str], optional*) – Alternative character termination to use. If None, the value of write_termination is used. Defaults to None.
- **encoding** (*Optional[str], optional*) – Alternative encoding to use to turn str into bytes. If None, the value of encoding is used. Defaults to None.

Returns Number of bytes written.

Return type int

write_ascii_values(*message: str, values: Sequence[Any], converter: Union[typing_extensions.Literal['s', 'b', 'c', 'd', 'o', 'x', 'X', 'e', 'E', 'f', 'F', 'g', 'G'][s, b, c, d, o, x, X, e, E, f, F, g, G], Callable[[str], Any]] = 'f', separator: Union[str, Callable[[Iterable[str]], str]] = ',', termination: Optional[str] = None, encoding: Optional[str] = None*)

Write a string message to the device followed by values in ascii format.

The write_termination is always appended to it.

Parameters

- **message** (*str*) – Header of the message to be sent.
- **values** (*Sequence[Any]*) – Data to be writen to the device.
- **converter** (*Union[str, Callable[[Any], str]], optional*) – Str formatting codes or function used to convert each value. Defaults to "f".

- **separator** (*Union[str, Callable[[Iterable[str]], str]], optional*) – Str or callable that join the values in a single str. If a str is given, separator.join(values) is used. Defaults to ','
- **termination** (*Optional[str], optional*) – Alternative character termination to use. If None, the value of write_termination is used. Defaults to None.
- **encoding** (*Optional[str], optional*) – Alternative encoding to use to turn str into bytes. If None, the value of encoding is used. Defaults to None.

Returns Number of bytes written.

Return type int

write_binary_values(*message: str, values: Sequence[Any], datatype: typing_extensions.Literal['s', 'b', 'B', 'h', 'H', 'i', 'I', 'l', 'L', 'q', 'Q', 'f', 'd'][s, b, B, h, H, i, I, l, L, q, Q, f, d] = 'f', is_big_endian: bool = False, termination: Optional[str] = None, encoding: Optional[str] = None, header_fmt: typing_extensions.Literal['ieee', 'hp', 'empty'][ieee, hp, empty] = 'ieee'*)

Write a string message to the device followed by values in binary format.

The write_termination is always appended to it.

Parameters

- **message** (*str*) – The header of the message to be sent.
- **values** (*Sequence[Any]*) – Data to be written to the device.
- **datatype** (*util.BINARY_DATATYPES, optional*) – The format string for a single element. See struct module.
- **is_big_endian** (*bool, optional*) – Are the data in big or little endian order.
- **termination** (*Optional[str], optional*) – Alternative character termination to use. If None, the value of write_termination is used. Defaults to None.
- **encoding** (*Optional[str], optional*) – Alternative encoding to use to turn str into bytes. If None, the value of encoding is used. Defaults to None.
- **header_fmt** (*util.BINARY_HEADERS*) – Format of the header prefixing the data.

Returns Number of bytes written.

Return type int

write_raw(*message: bytes*) → int

Write a byte message to the device.

Parameters **message** (*bytes*) – The message to be sent.

Returns Number of bytes written

Return type int

write_termination

Write termination character.

class pyvisa.resources.**TCPIPSocket**(*resource_manager: pyvisa.highlevel.ResourceManager, resource_name: str*)

Communicates with to devices of type TCPIP::host address::port::SOCKET

More complex resource names can be specified with the following grammar: TCPIP[board]::host address::port::SOCKET

Do not instantiate directly, use *pyvisa.highlevel.ResourceManager.open_resource()*.

CR = '\r'

LF = '\n'

allow_dma
Should I/O accesses use DMA (True) or Programmed I/O (False).

In some implementations, this attribute may have global effects even though it is documented to be a local attribute. Since this affects performance and not functionality, that behavior is acceptable.

VISA Attribute VI_ATTR_DMA_ALLOW_EN (1073676318)

Type bool

assert_trigger() → None
Sends a software trigger to the device.

before_close() → None
Called just before closing an instrument.

chunk_size = 20480

clear() → None
Clear this resource.

close() → None
Closes the VISA session and marks the handle as invalid.

disable_event(*event_type: pyvisa.constants.EventType*, *mechanism: pyvisa.constants.EventMechanism*) → None
Disable notification for an event type(s) via the specified mechanism(s).

Parameters

- **event_type** (*constants.EventType*) – Logical event identifier.
- **mechanism** (*constants.EventMechanism*) – Specifies event handling mechanisms to be disabled.

discard_events(*event_type: pyvisa.constants.EventType*, *mechanism: pyvisa.constants.EventMechanism*) → None
Discards event occurrences for an event type and mechanism in this resource.

Parameters

- **event_type** (*constants.EventType*) – Logical event identifier.
- **mechanism** (*constants.EventMechanism*) – Specifies event handling mechanisms to be disabled.

enable_event(*event_type: pyvisa.constants.EventType*, *mechanism: pyvisa.constants.EventMechanism*, *context: None = None*) → None
Enable event occurrences for specified event types and mechanisms in this resource.

Parameters

- **event_type** (*constants.EventType*) – Logical event identifier.
- **mechanism** (*constants.EventMechanism*) – Specifies event handling mechanisms to be enabled
- **context** (*None*) – Not currently used, leave as None.

encoding
Encoding used for read and write operations.

flush(*mask: pyvisa.constants.BufferOperation*) → None
Manually clears the specified buffers.

Depending on the value of the mask this can cause the buffer data to be written to the device.

Parameters **mask** (*constants.BufferOperation*) – Specifies the action to be taken with flushing the buffer. See highlevel.VisaLibraryBase.flush for a detailed description.

get_visa_attribute(*name: pyvisa.constants.ResourceAttribute*) → Any
Retrieves the state of an attribute in this resource.

One should prefer the dedicated descriptor for often used attributes since those perform checks and automatic conversion on the value.

Parameters **name** (*constants.ResourceAttribute*) – Resource attribute for which the state query is made.

Returns The state of the queried attribute for a specified resource.

Return type Any

ignore_warning(**warnings_constants*) → AbstractContextManager[T_co]
Ignoring warnings context manager for the current resource.

Parameters **warnings_constants** (constants.StatusCode) – Constants identifying the warnings to ignore.

implementation_version
Resource version that identifies the revisions or implementations of a resource.

This attribute value is defined by the individual manufacturer and increments with each new revision. The format of the value has the upper 12 bits as the major number of the version, the next lower 12 bits as the minor number of the version, and the lowest 8 bits as the sub-minor number of the version.

VISA Attribute VI_ATTR_RSRC_IMPL_VERSION (1073676291)

Type int

Range 0 <= value <= 4294967295

install_handler(*event_type: pyvisa.constants.EventType, handler: Callable[[NewType.<locals>.new_type, pyvisa.constants.EventType, NewType.<locals>.new_type, Any], None], user_handle=None*) → Any
Install handlers for event callbacks in this resource.

Parameters

- **event_type** (*constants.EventType*) – Logical event identifier.
- **handler** (*VISAHandler*) – Handler function to be installed by a client application.
- **user_handle** – A value specified by an application that can be used for identifying handlers uniquely for an event type. Depending on the backend they may be restriction on the possible values. Look at the backend *install_visa_handler* for more details.

Returns User handle in a format amenable to the backend. This is this representation of the handle that should be used when unistalling a handler.

Return type Any

interface_number
Board number for the given interface. :VISA Attribute: VI_ATTR_INTF_NUM (1073676662) :type: int :range: 0 <= value <= 65535

interface_type
Interface type of the given session. :VISA Attribute: VI_ATTR_INTF_TYPE (1073676657) :type: :class:pyvisa.constants.InterfaceType

io_protocol
IO protocol to use.

In VXI, you can choose normal word serial or fast data channel (FDC). In GPIB, you can choose normal or high-speed (HS-488) transfers. In serial, TCPIP, or USB RAW, you can choose normal transfers or 488.2-defined strings. In USB INSTR, you can choose normal or vendor-specific transfers.

VISA Attribute VI_ATTR_IO_PROT (1073676316)

Type :class:pyvisa.constants.IOProtocol

last_status
Last status code for this session.

lock(*timeout: Union[float, typing_extensions.Literal['default']][default]] = 'default', requested_key: Optional[str] = None*) → str
Establish a shared lock to the resource.

Parameters

- **timeout** (*Union[float, Literal["default"]], optional*) – Absolute time period (in milliseconds) that a resource waits to get unlocked by the locking session before returning an error. Defaults to "default" which means use self.timeout.
- **requested_key** (*Optional[str], optional*) – Access key used by another session with which you want your session to share a lock or None to generate a new shared access key.

Returns A new shared access key if requested_key is None, otherwise, same value as the requested_key

Return type str

lock_context(*timeout: Union[float, typing_extensions.Literal['default']][default]] = 'default', requested_key: Optional[str] = 'exclusive'*) → Iterator[Optional[str]]
A context that locks

Parameters

- **timeout** (*Union[float, Literal["default"]], optional*) – Absolute time period (in milliseconds) that a resource waits to get unlocked by the locking session before returning an error. Defaults to "default" which means use self.timeout.
- **requested_key** (*Optional[str], optional*) – When using default of 'exclusive' the lock is an exclusive lock. Otherwise it is the access key for the shared lock or None to generate a new shared access key.

Yields *Optional[str]* – The access_key if applicable.

lock_excl(*timeout: Union[float, typing_extensions.Literal['default']][default]] = 'default'*) → None
Establish an exclusive lock to the resource.

Parameters **timeout** (*Union[float, Literal["default"]], optional*) – Absolute time period (in milliseconds) that a resource waits to get unlocked by the locking session before returning an error. Defaults to "default" which means use self.timeout.

lock_state
Current locking state of the resource.

The resource can be unlocked, locked with an exclusive lock, or locked with a shared lock.

VISA Attribute VI_ATTR_RSRC_LOCK_STATE (1073676292)

Type :class:pyvisa.constants.AccessModes

open(*access_mode: pyvisa.constants.AccessModes = <AccessModes.no_lock: 0>, open_timeout: int = 5000*) → None
Opens a session to the specified resource.

Parameters

- **access_mode** (*constants.AccessModes, optional*) – Specifies the mode by which the resource is to be accessed. Defaults to constants.AccessModes.no_lock.
- **open_timeout** (*int, optional*) – If the access_mode parameter requests a lock, then this parameter specifies the absolute time period (in milliseconds) that the resource waits to get unlocked before this operation returns an error. Defaults to 5000.

query(*message: str, delay: Optional[float] = None*) → str
A combination of write(message) and read()

Parameters

- **message** (*str*) – The message to send.
- **delay** (*Optional[float], optional*) – Delay in seconds between write and read operations. If None, defaults to self.query_delay.

Returns Answer from the device.

Return type str

query_ascii_values(*message: str, converter: Union[typing_extensions.Literal['s', 'b', 'c', 'd', 'o', 'x', 'X', 'e', 'E', 'f', 'F', 'g', 'G'][s, b, c, d, o, x, X, e, E, f, F, g, G], Callable[[str], Any]] = 'f', separator: Union[str, Callable[[str], Iterable[str]]] = ',', container: Union[Type[CT_co], Callable[[Iterable[T_co]], Sequence[T_co]]] = <class 'list'>, delay: Optional[float] = None*) → Sequence[Any]
Query the device for values in ascii format returning an iterable of values.

Parameters

- **message** (*str*) – The message to send.
- **converter** (*ASCII_CONVERTER, optional*) – Str format of function to convert each value. Default to "f".
- **separator** (*Union[str, Callable[[str], Iterable[str]]]*) – str or callable used to split the data into individual elements. If a str is given, data.split(separator) is used. Default to ",".
- **container** (*Union[Type, Callable[[Iterable], Sequence]], optional*) – Container type to use for the output data. Possible values are: list, tuple, np.ndarray, etc, Default to list.

- **delay** (*Optional[float], optional*) – Delay in seconds between write and read operations. If None, defaults to self.query_delay.

Returns Parsed data.

Return type Sequence

query_binary_values(*message: str, datatype: typing_extensions.Literal['s', 'b', 'B', 'h', 'H', 'i', 'I', 'l', 'L', 'q', 'Q', 'f', 'd'][s, b, B, h, H, i, I, l, L, q, Q, f, d] = 'f', is_big_endian: bool = False, container: Union[Type[CT_co], Callable[[Iterable[T_co]], Sequence[T_co]]] = <class 'list'>, delay: Optional[float] = None, header_fmt: typing_extensions.Literal['ieee', 'hp', 'empty'][ieee, hp, empty] = 'ieee', expect_termination: bool = True, data_points: int = 0, chunk_size: Optional[int] = None*) → Sequence[Union[int, float]]

Query the device for values in binary format returning an iterable of values.

Parameters

- **message** (*str*) – The message to send.
- **datatype** (*BINARY_DATATYPES, optional*) – Format string for a single element. See struct module. 'f' by default.
- **is_big_endian** (*bool, optional*) – Are the data in big or little endian order. Defaults to False.
- **container** (*Union[Type, Callable[[Iterable], Sequence]], optional*) – Container type to use for the output data. Possible values are: list, tuple, np.ndarray, etc, Default to list.
- **delay** (*Optional[float], optional*) – Delay in seconds between write and read operations. If None, defaults to self.query_delay.
- **header_fmt** (*util.BINARY_HEADERS, optional*) – Format of the header prefixing the data. Defaults to 'ieee'.
- **expect_termination** (*bool, optional*) – When set to False, the expected length of the binary values block does not account for the final termination character (the read termination). Defaults to True.
- **data_points** (*int, optional*) – Number of points expected in the block. This is used only if the instrument does not report it itself. This will be converted in a number of bytes based on the datatype. Defaults to 0.
- **chunk_size** (*int, optional*) – Size of the chunks to read from the device. Using larger chunks may be faster for large amount of data.

Returns Data read from the device.

Return type Sequence[Union[int, float]]

query_delay = 0.0

read(*termination: Optional[str] = None, encoding: Optional[str] = None*) → str

Read a string from the device.

Reading stops when the device stops sending (e.g. by setting appropriate bus lines), or the termination characters sequence was detected. Attention: Only the last character of the termination characters is really used to stop reading, however, the whole sequence is compared to the ending of the read string message. If they don't match, a warning is issued.

Parameters

- **termination** (*Optional[str], optional*) – Alternative character termination to use. If None, the value of write_termination is used. Defaults to None.
- **encoding** (*Optional[str], optional*) – Alternative encoding to use to turn bytes into str. If None, the value of encoding is used. Defaults to None.

Returns Message read from the instrument and decoded.

Return type str

read_ascii_values(*converter: Union[typing_extensions.Literal['s', 'b', 'c', 'd', 'o', 'x', 'X', 'e', 'E', 'f', 'F', 'g', 'G'][s, b, c, d, o, x, X, e, E, f, F, g, G], Callable[[str], Any]] = 'f', separator: Union[str, Callable[[str], Iterable[str]]] = ',', container: Union[Type[CT_co], Callable[[Iterable[T_co]], Sequence[T_co]]] = <class 'list'>*) → Sequence[T_co]

Read values from the device in ascii format returning an iterable of values.

Parameters **converter** (*ASCII_CONVERTER, optional*) – Str format of function to convert each value. Default to "f".

separator [Union[str, Callable[[str], Iterable[str]]]] str or callable used to split the data into individual elements. If a str is given, data.split(separator) is used. Default to ",".

container [Union[Type, Callable[[Iterable], Sequence]], optional] Container type to use for the output data. Possible values are: list, tuple, np.ndarray, etc, Default to list.

Returns Parsed data.

Return type Sequence

read_binary_values(*datatype: typing_extensions.Literal['s', 'b', 'B', 'h', 'H', 'i', 'I', 'l', 'L', 'q', 'Q', 'f', 'd'][s, b, B, h, H, i, I, l, L, q, Q, f, d] = 'f', is_big_endian: bool = False, container: Union[Type[CT_co], Callable[[Iterable[T_co]], Sequence[T_co]]] = <class 'list'>, header_fmt: typing_extensions.Literal['ieee', 'hp', 'empty'][ieee, hp, empty] = 'ieee', expect_termination: bool = True, data_points: int = 0, chunk_size: Optional[int] = None*) → Sequence[Union[int, float]]

Read values from the device in binary format returning an iterable of values.

Parameters

- **datatype** (*BINARY_DATATYPES, optional*) – Format string for a single element. See struct module. 'f' by default.
- **is_big_endian** (*bool, optional*) – Are the data in big or little endian order. Defaults to False.
- **container** (*Union[Type, Callable[[Iterable], Sequence]], optional*) – Container type to use for the output data. Possible values are: list, tuple, np.ndarray, etc, Default to list.
- **header_fmt** (*util.BINARY_HEADERS, optional*) – Format of the header prefixing the data. Defaults to 'ieee'.
- **expect_termination** (*bool, optional*) – When set to False, the expected length of the binary values block does not account for the final termination character (the read termination). Defaults to True.
- **data_points** (*int, optional*) – Number of points expected in the block. This is used only if the instrument does not report it itself. This will be converted in a number of bytes based on the datatype. Defaults to 0.

- **chunk_size** (*int, optional*) – Size of the chunks to read from the device. Using larger chunks may be faster for large amount of data.

Returns Data read from the device.

Return type Sequence[Union[int, float]]

read_bytes(*count: int*, *chunk_size: Optional[int] = None*, *break_on_termchar: bool = False*) → bytes

Read a certain number of bytes from the instrument.

Parameters

- **count** (*int*) – The number of bytes to read from the instrument.
- **chunk_size** (*Optional[int], optional*) – The chunk size to use to perform the reading. If count > chunk_size multiple low level operations will be performed. Defaults to None, meaning the resource wide set value is set.
- **break_on_termchar** (*bool, optional*) – Should the reading stop when a termination character is encountered or when the message ends. Defaults to False.

Returns Bytes read from the instrument.

Return type bytes

read_raw(*size: Optional[int] = None*) → bytes

Read the unmodified string sent from the instrument to the computer.

In contrast to read(), no termination characters are stripped.

Parameters **size** (*Optional[int], optional*) – The chunk size to use to perform the reading. Defaults to None, meaning the resource wide set value is set.

Returns Bytes read from the instrument.

Return type bytes

read_stb() → int

Service request status register.

read_termination

Read termination character.

read_termination_context(*new_termination: str*) → Iterator[T_co]

classmethod register(*interface_type: pyvisa.constants.InterfaceType*, *resource_class: str*) → Callable[[Type[T]], Type[T]]

Create a decorator to register a class.

The class is associated to an interface type, resource class pair.

Parameters

- **interface_type** (constants.InterfaceType) – Interface type for which to register a wrapper class.
- **resource_class** (*str*) – Resource class for which to register a wrapper class.

Returns Decorator registering the class. Raises TypeError if some VISA attributes are missing on the registered class.

Return type Callable[[Type[T]], Type[T]]

resource_class

Resource class as defined by the canonical resource name.

Possible values are: INSTR, INTFC, SOCKET, RAW...

VISA Attribute VI_ATTR_RSRC_CLASS (3221159937)

resource_info
Get the extended information of this resource.

resource_manufacturer_name
Manufacturer name of the vendor that implemented the VISA library.

This attribute is not related to the device manufacturer attributes.

Note The value of this attribute is for display purposes only and not for programmatic decisions, as the value can differ between VISA implementations and/or revisions.

VISA Attribute VI_ATTR_RSRC_MANF_NAME (3221160308)

resource_name
Unique identifier for a resource compliant with the address structure. :VISA Attribute: VI_ATTR_RSRC_NAME (3221159938)

send_end
Should END be asserted during the transfer of the last byte of the buffer. :VISA Attribute: VI_ATTR_SEND_END_EN (1073676310) :type: bool

session
Resource session handle.

Raises `errors.InvalidSession` – Raised if session is closed.

set_visa_attribute(*name: pyvisa.constants.ResourceAttribute*, *state: Any*) → pyvisa.constants.StatusCode
Set the state of an attribute.

One should prefer the dedicated descriptor for often used attributes since those perform checks and automatic conversion on the value.

Parameters

- **name** (*constants.ResourceAttribute*) – Attribute for which the state is to be modified.
- **state** (*Any*) – The state of the attribute to be set for the specified object.

Returns Return value of the library call.

Return type *constants.StatusCode*

spec_version
Version of the VISA specification to which the implementation is compliant.

The format of the value has the upper 12 bits as the major number of the version, the next lower 12 bits as the minor number of the version, and the lowest 8 bits as the sub-minor number of the version. The current VISA specification defines the value to be 00300000h.

VISA Attribute VI_ATTR_RSRC_SPEC_VERSION (1073676656)

Type int

Range 0 <= value <= 4294967295

stb
Service request status register.

timeout
Timeout in milliseconds for all resource I/O operations.

This value is used when accessing the device associated with the given session.

Special values:

- **immediate (VI_TMO_IMMEDIATE): 0** (for convenience, any value smaller than 1 is considered as 0)
- **infinite (VI_TMO_INFINITE): float('+inf')** (for convenience, None is considered as float('+inf'))

To set an **infinite** timeout, you can also use:

```
>>> del instrument.timeout
```

A timeout value of VI_TMO_IMMEDIATE means that operations should never wait for the device to respond. A timeout value of VI_TMO_INFINITE disables the timeout mechanism.

VISA Attribute VI_ATTR_TMO_VALUE (1073676314)

Type int

Range 0 <= value <= 4294967295

uninstall_handler(*event_type: pyvisa.constants.EventType, handler: Callable[[NewType.<locals>.new_type, pyvisa.constants.EventType, NewType.<locals>.new_type, Any], None], user_handle=None*) → None
Uninstalls handlers for events in this resource.

Parameters

- **event_type** (*constants.EventType*) – Logical event identifier.
- **handler** (*VISAHandler*) – Handler function to be uninstalled by a client application.
- **user_handle** (*Any*) – The user handle returned by install_handler.

unlock() → None
Relinquishes a lock for the specified resource.

visa_attributes_classes = {<class 'pyvisa.attributes.AttrVI_ATTR_TCPIP_PORT'>, <class

wait_on_event(*in_event_type: pyvisa.constants.EventType, timeout: int, capture_timeout: bool = False*) → pyvisa.resources.resource.WaitResponse
Waits for an occurrence of the specified event in this resource.

in_event_type [constants.EventType] Logical identifier of the event(s) to wait for.

timeout [int] Absolute time period in time units that the resource shall wait for a specified event to occur before returning the time elapsed error. The time unit is in milliseconds. None means waiting forever if necessary.

capture_timeout [bool, optional] When True will not produce a VisaIOError(VI_ERROR_TMO) but instead return a WaitResponse with timed_out=True.

Returns Object that contains event_type, context and ret value.

Return type WaitResponse

wrap_handler(*callable: Callable[[Resource, pyvisa.events.Event, Any], None]*) → Callable[[NewType.<locals>.new_type, pyvisa.constants.EventType, NewType.<locals>.new_type, Any], None]

Wrap an event handler to provide the signature expected by VISA.

The handler is expected to have the following signature: handler(resource: Resource, event: Event, user_handle: Any) -> None.

The wrapped handler should be used only to handle events on the resource used to wrap the handler.

write(*message: str*, *termination: Optional[str] = None*, *encoding: Optional[str] = None*) → int

Write a string message to the device.

The write_termination is always appended to it.

Parameters

- **message** (*str*) – The message to be sent.
- **termination** (*Optional[str], optional*) – Alternative character termination to use. If None, the value of write_termination is used. Defaults to None.
- **encoding** (*Optional[str], optional*) – Alternative encoding to use to turn str into bytes. If None, the value of encoding is used. Defaults to None.

Returns Number of bytes written.

Return type int

write_ascii_values(*message: str, values: Sequence[Any], converter: Union[typing_extensions.Literal['s', 'b', 'c', 'd', 'o', 'x', 'X', 'e', 'E', 'f', 'F', 'g', 'G'][s, b, c, d, o, x, X, e, E, f, F, g, G], Callable[[str], Any]] = 'f', separator: Union[str, Callable[[Iterable[str]], str]] = ',', termination: Optional[str] = None, encoding: Optional[str] = None*)

Write a string message to the device followed by values in ascii format.

The write_termination is always appended to it.

Parameters

- **message** (*str*) – Header of the message to be sent.
- **values** (*Sequence[Any]*) – Data to be writen to the device.
- **converter** (*Union[str, Callable[[Any], str]], optional*) – Str formatting codes or function used to convert each value. Defaults to "f".
- **separator** (*Union[str, Callable[[Iterable[str]], str]], optional*) – Str or callable that join the values in a single str. If a str is given, separator.join(values) is used. Defaults to ','
- **termination** (*Optional[str], optional*) – Alternative character termination to use. If None, the value of write_termination is used. Defaults to None.
- **encoding** (*Optional[str], optional*) – Alternative encoding to use to turn str into bytes. If None, the value of encoding is used. Defaults to None.

Returns Number of bytes written.

Return type int

write_binary_values(*message: str, values: Sequence[Any], datatype: typing_extensions.Literal['s', 'b', 'B', 'h', 'H', 'i', 'I', 'l', 'L', 'q', 'Q', 'f', 'd'][s, b, B, h, H, i, I, l, L, q, Q, f, d] = 'f', is_big_endian: bool = False, termination: Optional[str] = None, encoding: Optional[str] = None, header_fmt: typing_extensions.Literal['ieee', 'hp', 'empty'][ieee, hp, empty] = 'ieee'*)

Write a string message to the device followed by values in binary format.

The write_termination is always appended to it.

Parameters

- **message** (*str*) – The header of the message to be sent.
- **values** (*Sequence[Any]*) – Data to be written to the device.
- **datatype** (*util.BINARY_DATATYPES, optional*) – The format string for a single element. See struct module.
- **is_big_endian** (*bool, optional*) – Are the data in big or little endian order.
- **termination** (*Optional[str], optional*) – Alternative character termination to use. If None, the value of write_termination is used. Defaults to None.
- **encoding** (*Optional[str], optional*) – Alternative encoding to use to turn str into bytes. If None, the value of encoding is used. Defaults to None.
- **header_fmt** (*util.BINARY_HEADERS*) – Format of the header prefixing the data.

Returns Number of bytes written.

Return type int

write_raw(*message: bytes*) → int

Write a byte message to the device.

Parameters **message** (*bytes*) – The message to be sent.

Returns Number of bytes written

Return type int

write_termination

Write termination character.

class pyvisa.resources.**USBInstrument**(*resource_manager: pyvisa.highlevel.ResourceManager, resource_name: str*)

USB INSTR resources USB::manufacturer ID::model code::serial number

More complex resource names can be specified with the following grammar: USB[board]::manufacturer ID::model code::serial number[::USB interface number][::INSTR]

Do not instantiate directly, use *pyvisa.highlevel.ResourceManager.open_resource()*.

CR = '\r'

LF = '\n'

allow_dma

Should I/O accesses use DMA (True) or Programmed I/O (False).

In some implementations, this attribute may have global effects even though it is documented to be a local attribute. Since this affects performance and not functionality, that behavior is acceptable.

VISA Attribute VI_ATTR_DMA_ALLOW_EN (1073676318)

Type bool

assert_trigger() → None
Sends a software trigger to the device.

before_close() → None
Called just before closing an instrument.

chunk_size = 20480

clear() → None
Clear this resource.

close() → None
Closes the VISA session and marks the handle as invalid.

control_in(*request_type_bitmap_field: int*, *request_id: int*, *request_value: int*, *index: int*, *length: int = 0*) → bytes
Performs a USB control pipe transfer from the device.

Parameters

- **request_type_bitmap_field** (*int*) – bmRequestType parameter of the setup stage of a USB control transfer.
- **request_id** (*int*) – bRequest parameter of the setup stage of a USB control transfer.
- **request_value** (*int*) – wValue parameter of the setup stage of a USB control transfer.
- **index** (*int*) – wIndex parameter of the setup stage of a USB control transfer. This is usually the index of the interface or endpoint.
- **length** (*int*) – wLength parameter of the setup stage of a USB control transfer. This value also specifies the size of the data buffer to receive the data from the optional data stage of the control transfer.

Returns The data buffer that receives the data from the optional data stage of the control transfer.

Return type bytes

control_out(*request_type_bitmap_field: int*, *request_id: int*, *request_value: int*, *index: int*, *data: bytes = b''*)
Performs a USB control pipe transfer to the device.

Parameters

- **request_type_bitmap_field** (*int*) – bmRequestType parameter of the setup stage of a USB control transfer.
- **request_id** (*int*) – bRequest parameter of the setup stage of a USB control transfer.
- **request_value** (*int*) – wValue parameter of the setup stage of a USB control transfer.
- **index** (*int*) – wIndex parameter of the setup stage of a USB control transfer. This is usually the index of the interface or endpoint.
- **data** (*str*) – The data buffer that sends the data in the optional data stage of the control transfer.

control_ren(*mode: pyvisa.constants.RENLineOperation*) → pyvisa.constants.StatusCode
Controls the state of the GPIB Remote Enable (REN) interface line.

The remote/local state of the device can also be controlled optionally.

Corresponds to viGpibControlREN function of the VISA library.

Parameters **mode** (*constants.RENLineOperation*) – Specifies the state of the REN line and optionally the device remote/local state.

Returns Return value of the library call.

Return type *constants.StatusCode*

disable_event(*event_type: pyvisa.constants.EventType*, *mechanism: pyvisa.constants.EventMechanism*) → None
Disable notification for an event type(s) via the specified mechanism(s).

Parameters

- **event_type** (*constants.EventType*) – Logical event identifier.
- **mechanism** (*constants.EventMechanism*) – Specifies event handling mechanisms to be disabled.

discard_events(*event_type: pyvisa.constants.EventType*, *mechanism: pyvisa.constants.EventMechanism*) → None
Discards event occurrences for an event type and mechanism in this resource.

Parameters

- **event_type** (*constants.EventType*) – Logical event identifier.
- **mechanism** (*constants.EventMechanism*) – Specifies event handling mechanisms to be disabled.

enable_event(*event_type: pyvisa.constants.EventType*, *mechanism: pyvisa.constants.EventMechanism*, *context: None = None*) → None
Enable event occurrences for specified event types and mechanisms in this resource.

Parameters

- **event_type** (*constants.EventType*) – Logical event identifier.
- **mechanism** (*constants.EventMechanism*) – Specifies event handling mechanisms to be enabled
- **context** (*None*) – Not currently used, leave as None.

encoding
Encoding used for read and write operations.

flush(*mask: pyvisa.constants.BufferOperation*) → None
Manually clears the specified buffers.

Depending on the value of the mask this can cause the buffer data to be written to the device.

Parameters **mask** (*constants.BufferOperation*) – Specifies the action to be taken with flushing the buffer. See highlevel.VisaLibraryBase.flush for a detailed description.

get_visa_attribute(*name: pyvisa.constants.ResourceAttribute*) → Any
Retrieves the state of an attribute in this resource.

One should prefer the dedicated descriptor for often used attributes since those perform checks and automatic conversion on the value.

Parameters **name** (*constants.ResourceAttribute*) – Resource attribute for which the state query is made.

Returns The state of the queried attribute for a specified resource.

Return type Any

ignore_warning(**warnings_constants*) → AbstractContextManager[T_co]
Ignoring warnings context manager for the current resource.

Parameters **warnings_constants** (constants.StatusCode) – Constants identifying the warnings to ignore.

implementation_version
Resource version that identifies the revisions or implementations of a resource.

This attribute value is defined by the individual manufacturer and increments with each new revision. The format of the value has the upper 12 bits as the major number of the version, the next lower 12 bits as the minor number of the version, and the lowest 8 bits as the sub-minor number of the version.

VISA Attribute VI_ATTR_RSRC_IMPL_VERSION (1073676291)

Type int

Range 0 <= value <= 4294967295

install_handler(*event_type: pyvisa.constants.EventType, handler: Callable[[NewType.<locals>.new_type, pyvisa.constants.EventType, NewType.<locals>.new_type, Any], None], user_handle=None*) → Any
Install handlers for event callbacks in this resource.

Parameters

- **event_type** (*constants.EventType*) – Logical event identifier.
- **handler** (*VISAHandler*) – Handler function to be installed by a client application.
- **user_handle** – A value specified by an application that can be used for identifying handlers uniquely for an event type. Depending on the backend they may be restriction on the possible values. Look at the backend *install_visa_handler* for more details.

Returns User handle in a format amenable to the backend. This is this representation of the handle that should be used when unistalling a handler.

Return type Any

interface_number
USB interface number used by the given session. :VISA Attribute: VI_ATTR_USB_INTFC_NUM (1073676705) :type: int :range: 0 <= value <= 254

interface_type
Interface type of the given session. :VISA Attribute: VI_ATTR_INTF_TYPE (1073676657) :type: :class:pyvisa.constants.InterfaceType

io_protocol
IO protocol to use.

In VXI, you can choose normal word serial or fast data channel (FDC). In GPIB, you can choose normal or high-speed (HS-488) transfers. In serial, TCPIP, or USB RAW, you can choose normal transfers or 488.2-defined strings. In USB INSTR, you can choose normal or vendor-specific transfers.

VISA Attribute VI_ATTR_IO_PROT (1073676316)

Type :class:pyvisa.constants.IOProtocol

is_4882_compliant
Whether the device is 488.2 compliant.

last_status
Last status code for this session.

lock(*timeout: Union[float, typing_extensions.Literal['default']][default]] = 'default', requested_key: Optional[str] = None*) → str
Establish a shared lock to the resource.

Parameters

- **timeout** (*Union[float, Literal["default"]], optional*) – Absolute time period (in milliseconds) that a resource waits to get unlocked by the locking session before returning an error. Defaults to "default" which means use self.timeout.
- **requested_key** (*Optional[str], optional*) – Access key used by another session with which you want your session to share a lock or None to generate a new shared access key.

Returns A new shared access key if requested_key is None, otherwise, same value as the requested_key

Return type str

lock_context(*timeout: Union[float, typing_extensions.Literal['default']][default]] = 'default', requested_key: Optional[str] = 'exclusive'*) → Iterator[Optional[str]]
A context that locks

Parameters

- **timeout** (*Union[float, Literal["default"]], optional*) – Absolute time period (in milliseconds) that a resource waits to get unlocked by the locking session before returning an error. Defaults to "default" which means use self.timeout.
- **requested_key** (*Optional[str], optional*) – When using default of 'exclusive' the lock is an exclusive lock. Otherwise it is the access key for the shared lock or None to generate a new shared access key.

Yields *Optional[str]* – The access_key if applicable.

lock_excl(*timeout: Union[float, typing_extensions.Literal['default']][default]] = 'default'*) → None
Establish an exclusive lock to the resource.

Parameters **timeout** (*Union[float, Literal["default"]], optional*) – Absolute time period (in milliseconds) that a resource waits to get unlocked by the locking session before returning an error. Defaults to "default" which means use self.timeout.

lock_state
Current locking state of the resource.

The resource can be unlocked, locked with an exclusive lock, or locked with a shared lock.

VISA Attribute VI_ATTR_RSRC_LOCK_STATE (1073676292)

Type :class:pyvisa.constants.AccessModes

manufacturer_id
Manufacturer identification number of the device. :VISA Attribute: VI_ATTR_MANF_ID (1073676505) :type: int :range: 0 <= value <= 65535

manufacturer_name
Manufacturer name. :VISA Attribute: VI_ATTR_MANF_NAME (3221160050)

maximum_interrupt_size
Maximum size of data that will be stored by any given USB interrupt.

If a USB interrupt contains more data than this size, the data in excess of this size will be lost.

VISA Attribute VI_ATTR_USB_MAX_INTR_SIZE (1073676719)

Type int

Range 0 <= value <= 65535

model_code
Model code for the device. :VISA Attribute: VI_ATTR_MODEL_CODE (1073676511) :type: int :range: 0 <= value <= 65535

model_name
Model name of the device. :VISA Attribute: VI_ATTR_MODEL_NAME (3221160055)

open(*access_mode: pyvisa.constants.AccessModes = <AccessModes.no_lock: 0>*, *open_timeout: int = 5000*) → None
Opens a session to the specified resource.

Parameters

- **access_mode** (*constants.AccessModes, optional*) – Specifies the mode by which the resource is to be accessed. Defaults to constants.AccessModes.no_lock.
- **open_timeout** (*int, optional*) – If the access_mode parameter requests a lock, then this parameter specifies the absolute time period (in milliseconds) that the resource waits to get unlocked before this operation returns an error. Defaults to 5000.

query(*message: str*, *delay: Optional[float] = None*) → str
A combination of write(message) and read()

Parameters

- **message** (*str*) – The message to send.
- **delay** (*Optional[float], optional*) – Delay in seconds between write and read operations. If None, defaults to self.query_delay.

Returns Answer from the device.

Return type str

query_ascii_values(*message: str, converter: Union[typing_extensions.Literal['s', 'b', 'c', 'd', 'o', 'x', 'X', 'e', 'E', 'f', 'F', 'g', 'G'][s, b, c, d, o, x, X, e, E, f, F, g, G], Callable[[str], Any]] = 'f', separator: Union[str, Callable[[str], Iterable[str]]] = ',', container: Union[Type[CT_co], Callable[[Iterable[T_co]], Sequence[T_co]]] = <class 'list'>, delay: Optional[float] = None*) → Sequence[Any]
Query the device for values in ascii format returning an iterable of values.

Parameters

- **message** (*str*) – The message to send.
- **converter** (*ASCII_CONVERTER, optional*) – Str format of function to convert each value. Default to "f".
- **separator** (*Union[str, Callable[[str], Iterable[str]]]*) – str or callable used to split the data into individual elements. If a str is given, data.split(separator) is used. Default to ",".
- **container** (*Union[Type, Callable[[Iterable], Sequence]], optional*) – Container type to use for the output data. Possible values are: list, tuple, np.ndarray, etc, Default to list.

- **delay** (*Optional[float], optional*) – Delay in seconds between write and read operations. If None, defaults to self.query_delay.

Returns Parsed data.

Return type Sequence

query_binary_values(*message: str, datatype: typing_extensions.Literal['s', 'b', 'B', 'h', 'H', 'i', 'I', 'l', 'L', 'q', 'Q', 'f', 'd'][s, b, B, h, H, i, I, l, L, q, Q, f, d] = 'f', is_big_endian: bool = False, container: Union[Type[CT_co], Callable[[Iterable[T_co]], Sequence[T_co]]] = <class 'list'>, delay: Optional[float] = None, header_fmt: typing_extensions.Literal['ieee', 'hp', 'empty'][ieee, hp, empty] = 'ieee', expect_termination: bool = True, data_points: int = 0, chunk_size: Optional[int] = None*) → Sequence[Union[int, float]]

Query the device for values in binary format returning an iterable of values.

Parameters

- **message** (*str*) – The message to send.
- **datatype** (*BINARY_DATATYPES, optional*) – Format string for a single element. See struct module. 'f' by default.
- **is_big_endian** (*bool, optional*) – Are the data in big or little endian order. Defaults to False.
- **container** (*Union[Type, Callable[[Iterable], Sequence]], optional*) – Container type to use for the output data. Possible values are: list, tuple, np.ndarray, etc, Default to list.
- **delay** (*Optional[float], optional*) – Delay in seconds between write and read operations. If None, defaults to self.query_delay.
- **header_fmt** (*util.BINARY_HEADERS, optional*) – Format of the header prefixing the data. Defaults to 'ieee'.
- **expect_termination** (*bool, optional*) – When set to False, the expected length of the binary values block does not account for the final termination character (the read termination). Defaults to True.
- **data_points** (*int, optional*) – Number of points expected in the block. This is used only if the instrument does not report it itself. This will be converted in a number of bytes based on the datatype. Defaults to 0.
- **chunk_size** (*int, optional*) – Size of the chunks to read from the device. Using larger chunks may be faster for large amount of data.

Returns Data read from the device.

Return type Sequence[Union[int, float]]

query_delay = 0.0

read(*termination: Optional[str] = None, encoding: Optional[str] = None*) → str

Read a string from the device.

Reading stops when the device stops sending (e.g. by setting appropriate bus lines), or the termination characters sequence was detected. Attention: Only the last character of the termination characters is really used to stop reading, however, the whole sequence is compared to the ending of the read string message. If they don't match, a warning is issued.

Parameters

- **termination** (*Optional[str], optional*) – Alternative character termination to use. If None, the value of write_termination is used. Defaults to None.
- **encoding** (*Optional[str], optional*) – Alternative encoding to use to turn bytes into str. If None, the value of encoding is used. Defaults to None.

Returns Message read from the instrument and decoded.

Return type str

read_ascii_values(*converter: Union[typing_extensions.Literal['s', 'b', 'c', 'd', 'o', 'x', 'X', 'e', 'E', 'f', 'F', 'g', 'G'][s, b, c, d, o, x, X, e, E, f, F, g, G], Callable[[str], Any]] = 'f', separator: Union[str, Callable[[str], Iterable[str]]] = ',', container: Union[Type[CT_co], Callable[[Iterable[T_co]], Sequence[T_co]]] = <class 'list'>*) → Sequence[T_co]

Read values from the device in ascii format returning an iterable of values.

Parameters **converter** (*ASCII_CONVERTER, optional*) – Str format of function to convert each value. Default to "f".

separator [Union[str, Callable[[str], Iterable[str]]]] str or callable used to split the data into individual elements. If a str is given, data.split(separator) is used. Default to ",".

container [Union[Type, Callable[[Iterable], Sequence]], optional] Container type to use for the output data. Possible values are: list, tuple, np.ndarray, etc, Default to list.

Returns Parsed data.

Return type Sequence

read_binary_values(*datatype: typing_extensions.Literal['s', 'b', 'B', 'h', 'H', 'i', 'I', 'l', 'L', 'q', 'Q', 'f', 'd'][s, b, B, h, H, i, I, l, L, q, Q, f, d] = 'f', is_big_endian: bool = False, container: Union[Type[CT_co], Callable[[Iterable[T_co]], Sequence[T_co]]] = <class 'list'>, header_fmt: typing_extensions.Literal['ieee', 'hp', 'empty'][ieee, hp, empty] = 'ieee', expect_termination: bool = True, data_points: int = 0, chunk_size: Optional[int] = None*) → Sequence[Union[int, float]]

Read values from the device in binary format returning an iterable of values.

Parameters

- **datatype** (*BINARY_DATATYPES, optional*) – Format string for a single element. See struct module. 'f' by default.
- **is_big_endian** (*bool, optional*) – Are the data in big or little endian order. Defaults to False.
- **container** (*Union[Type, Callable[[Iterable], Sequence]], optional*) – Container type to use for the output data. Possible values are: list, tuple, np.ndarray, etc, Default to list.
- **header_fmt** (*util.BINARY_HEADERS, optional*) – Format of the header prefixing the data. Defaults to 'ieee'.
- **expect_termination** (*bool, optional*) – When set to False, the expected length of the binary values block does not account for the final termination character (the read termination). Defaults to True.
- **data_points** (*int, optional*) – Number of points expected in the block. This is used only if the instrument does not report it itself. This will be converted in a number of bytes based on the datatype. Defaults to 0.

- **chunk_size** (*int, optional*) – Size of the chunks to read from the device. Using larger chunks may be faster for large amount of data.

Returns Data read from the device.

Return type Sequence[Union[int, float]]

read_bytes(*count: int*, *chunk_size: Optional[int] = None*, *break_on_termchar: bool = False*) → bytes

Read a certain number of bytes from the instrument.

Parameters

- **count** (*int*) – The number of bytes to read from the instrument.
- **chunk_size** (*Optional[int], optional*) – The chunk size to use to perform the reading. If count > chunk_size multiple low level operations will be performed. Defaults to None, meaning the resource wide set value is set.
- **break_on_termchar** (*bool, optional*) – Should the reading stop when a termination character is encountered or when the message ends. Defaults to False.

Returns Bytes read from the instrument.

Return type bytes

read_raw(*size: Optional[int] = None*) → bytes

Read the unmodified string sent from the instrument to the computer.

In contrast to read(), no termination characters are stripped.

Parameters **size** (*Optional[int], optional*) – The chunk size to use to perform the reading. Defaults to None, meaning the resource wide set value is set.

Returns Bytes read from the instrument.

Return type bytes

read_stb() → int

Service request status register.

read_termination

Read termination character.

read_termination_context(*new_termination: str*) → Iterator[T_co]

classmethod register(*interface_type: pyvisa.constants.InterfaceType*, *resource_class: str*) → Callable[[Type[T]], Type[T]]

Create a decorator to register a class.

The class is associated to an interface type, resource class pair.

Parameters

- **interface_type** (constants.InterfaceType) – Interface type for which to register a wrapper class.
- **resource_class** (*str*) – Resource class for which to register a wrapper class.

Returns Decorator registering the class. Raises TypeError if some VISA attributes are missing on the registered class.

Return type Callable[[Type[T]], Type[T]]

resource_class

Resource class as defined by the canonical resource name.

Possible values are: INSTR, INTFC, SOCKET, RAW...

VISA Attribute VI_ATTR_RSRC_CLASS (3221159937)

resource_info
Get the extended information of this resource.

resource_manufacturer_name
Manufacturer name of the vendor that implemented the VISA library.

This attribute is not related to the device manufacturer attributes.

Note The value of this attribute is for display purposes only and not for programmatic decisions, as the value can differ between VISA implementations and/or revisions.

VISA Attribute VI_ATTR_RSRC_MANF_NAME (3221160308)

resource_name
Unique identifier for a resource compliant with the address structure. :VISA Attribute: VI_ATTR_RSRC_NAME (3221159938)

send_end
Should END be asserted during the transfer of the last byte of the buffer. :VISA Attribute: VI_ATTR_SEND_END_EN (1073676310) :type: bool

serial_number
USB serial number of this device. :VISA Attribute: VI_ATTR_USB_SERIAL_NUM (3221160352)

session
Resource session handle.

Raises `errors.InvalidSession` – Raised if session is closed.

set_visa_attribute(*name: pyvisa.constants.ResourceAttribute*, *state: Any*) → pyvisa.constants.StatusCode
Set the state of an attribute.

One should prefer the dedicated descriptor for often used attributes since those perform checks and automatic conversion on the value.

Parameters

- **name** (*constants.ResourceAttribute*) – Attribute for which the state is to be modified.
- **state** (*Any*) – The state of the attribute to be set for the specified object.

Returns Return value of the library call.

Return type *constants.StatusCode*

spec_version
Version of the VISA specification to which the implementation is compliant.

The format of the value has the upper 12 bits as the major number of the version, the next lower 12 bits as the minor number of the version, and the lowest 8 bits as the sub-minor number of the version. The current VISA specification defines the value to be 00300000h.

VISA Attribute VI_ATTR_RSRC_SPEC_VERSION (1073676656)

Type int

Range 0 <= value <= 4294967295

stb
Service request status register.

timeout
Timeout in milliseconds for all resource I/O operations.

This value is used when accessing the device associated with the given session.

Special values:

- **immediate (VI_TMO_IMMEDIATE): 0** (for convenience, any value smaller than 1 is considered as 0)
- **infinite (VI_TMO_INFINITE): float('+inf')** (for convenience, None is considered as float('+inf'))

To set an **infinite** timeout, you can also use:

```
>>> del instrument.timeout
```

A timeout value of VI_TMO_IMMEDIATE means that operations should never wait for the device to respond. A timeout value of VI_TMO_INFINITE disables the timeout mechanism.

VISA Attribute VI_ATTR_TMO_VALUE (1073676314)

Type int

Range 0 <= value <= 4294967295

uninstall_handler(*event_type: pyvisa.constants.EventType, handler: Callable[[NewType.<locals>.new_type, pyvisa.constants.EventType, NewType.<locals>.new_type, Any], None], user_handle=None*) → None
Uninstalls handlers for events in this resource.

Parameters

- **event_type** (*constants.EventType*) – Logical event identifier.
- **handler** (*VISAHandler*) – Handler function to be uninstalled by a client application.
- **user_handle** (*Any*) – The user handle returned by install_handler.

unlock() → None
Relinquishes a lock for the specified resource.

usb_protocol
USB protocol used by this USB interface. :VISA Attribute: VI_ATTR_USB_PROTOCOL (1073676711) :type: int :range: 0 <= value <= 255

visa_attributes_classes = {<class 'pyvisa.attributes.AttrVI_ATTR_INTF_INST_NAME'>, <cl

wait_on_event(*in_event_type: pyvisa.constants.EventType, timeout: int, capture_timeout: bool = False*) → pyvisa.resources.resource.WaitResponse
Waits for an occurrence of the specified event in this resource.

in_event_type [constants.EventType] Logical identifier of the event(s) to wait for.

timeout [int] Absolute time period in time units that the resource shall wait for a specified event to occur before returning the time elapsed error. The time unit is in milliseconds. None means waiting forever if necessary.

capture_timeout [bool, optional] When True will not produce a VisaIOError(VI_ERROR_TMO) but instead return a WaitResponse with timed_out=True.

Returns Object that contains event_type, context and ret value.

Return type WaitResponse

wrap_handler(*callable: Callable[[Resource, pyvisa.events.Event, Any], None]*) → Callable[[NewType.<locals>.new_type, pyvisa.constants.EventType, NewType.<locals>.new_type, Any], None]

Wrap an event handler to provide the signature expected by VISA.

The handler is expected to have the following signature: handler(resource: Resource, event: Event, user_handle: Any) -> None.

The wrapped handler should be used only to handle events on the resource used to wrap the handler.

write(*message: str*, *termination: Optional[str] = None*, *encoding: Optional[str] = None*) → int

Write a string message to the device.

The write_termination is always appended to it.

Parameters

- **message** (*str*) – The message to be sent.
- **termination** (*Optional[str], optional*) – Alternative character termination to use. If None, the value of write_termination is used. Defaults to None.
- **encoding** (*Optional[str], optional*) – Alternative encoding to use to turn str into bytes. If None, the value of encoding is used. Defaults to None.

Returns Number of bytes written.

Return type int

write_ascii_values(*message: str, values: Sequence[Any], converter: Union[typing_extensions.Literal['s', 'b', 'c', 'd', 'o', 'x', 'X', 'e', 'E', 'f', 'F', 'g', 'G'][s, b, c, d, o, x, X, e, E, f, F, g, G], Callable[[str], Any]] = 'f', separator: Union[str, Callable[[Iterable[str]], str]] = ',', termination: Optional[str] = None, encoding: Optional[str] = None*)

Write a string message to the device followed by values in ascii format.

The write_termination is always appended to it.

Parameters

- **message** (*str*) – Header of the message to be sent.
- **values** (*Sequence[Any]*) – Data to be writen to the device.
- **converter** (*Union[str, Callable[[Any], str]], optional*) – Str formatting codes or function used to convert each value. Defaults to "f".
- **separator** (*Union[str, Callable[[Iterable[str]], str]], optional*) – Str or callable that join the values in a single str. If a str is given, separator.join(values) is used. Defaults to ','
- **termination** (*Optional[str], optional*) – Alternative character termination to use. If None, the value of write_termination is used. Defaults to None.
- **encoding** (*Optional[str], optional*) – Alternative encoding to use to turn str into bytes. If None, the value of encoding is used. Defaults to None.

Returns Number of bytes written.

Return type int

write_binary_values(*message: str, values: Sequence[Any], datatype: typing_extensions.Literal['s', 'b', 'B', 'h', 'H', 'i', 'I', 'l', 'L', 'q', 'Q', 'f', 'd'][s, b, B, h, H, i, I, l, L, q, Q, f, d] = 'f', is_big_endian: bool = False, termination: Optional[str] = None, encoding: Optional[str] = None, header_fmt: typing_extensions.Literal['ieee', 'hp', 'empty'][ieee, hp, empty] = 'ieee'*)

Write a string message to the device followed by values in binary format.

The write_termination is always appended to it.

Parameters

- **message** (*str*) – The header of the message to be sent.
- **values** (*Sequence[Any]*) – Data to be written to the device.
- **datatype** (*util.BINARY_DATATYPES, optional*) – The format string for a single element. See struct module.
- **is_big_endian** (*bool, optional*) – Are the data in big or little endian order.
- **termination** (*Optional[str], optional*) – Alternative character termination to use. If None, the value of write_termination is used. Defaults to None.
- **encoding** (*Optional[str], optional*) – Alternative encoding to use to turn str into bytes. If None, the value of encoding is used. Defaults to None.
- **header_fmt** (*util.BINARY_HEADERS*) – Format of the header prefixing the data.

Returns Number of bytes written.

Return type int

write_raw(*message: bytes*) → int

Write a byte message to the device.

Parameters **message** (*bytes*) – The message to be sent.

Returns Number of bytes written

Return type int

write_termination

Write termination character.

class pyvisa.resources.**USBRaw**(*resource_manager: pyvisa.highlevel.ResourceManager, resource_name: str*)

USB RAW resources: USB::manufacturer ID::model code::serial number::RAW

More complex resource names can be specified with the following grammar: USB[board]::manufacturer ID::model code::serial number[::USB interface number]::RAW

Do not instantiate directly, use *pyvisa.highlevel.ResourceManager.open_resource()*.

CR = '\r'

LF = '\n'

allow_dma

Should I/O accesses use DMA (True) or Programmed I/O (False).

In some implementations, this attribute may have global effects even though it is documented to be a local attribute. Since this affects performance and not functionality, that behavior is acceptable.

VISA Attribute VI_ATTR_DMA_ALLOW_EN (1073676318)

Type bool

assert_trigger() → None
Sends a software trigger to the device.

before_close() → None
Called just before closing an instrument.

chunk_size = 20480

clear() → None
Clear this resource.

close() → None
Closes the VISA session and marks the handle as invalid.

disable_event(*event_type: pyvisa.constants.EventType*, *mechanism: pyvisa.constants.EventMechanism*) → None
Disable notification for an event type(s) via the specified mechanism(s).

Parameters

- **event_type** (*constants.EventType*) – Logical event identifier.
- **mechanism** (*constants.EventMechanism*) – Specifies event handling mechanisms to be disabled.

discard_events(*event_type: pyvisa.constants.EventType*, *mechanism: pyvisa.constants.EventMechanism*) → None
Discards event occurrences for an event type and mechanism in this resource.

Parameters

- **event_type** (*constants.EventType*) – Logical event identifier.
- **mechanism** (*constants.EventMechanism*) – Specifies event handling mechanisms to be disabled.

enable_event(*event_type: pyvisa.constants.EventType*, *mechanism: pyvisa.constants.EventMechanism*, *context: None = None*) → None
Enable event occurrences for specified event types and mechanisms in this resource.

Parameters

- **event_type** (*constants.EventType*) – Logical event identifier.
- **mechanism** (*constants.EventMechanism*) – Specifies event handling mechanisms to be enabled
- **context** (*None*) – Not currently used, leave as None.

encoding
Encoding used for read and write operations.

flush(*mask: pyvisa.constants.BufferOperation*) → None
Manually clears the specified buffers.

Depending on the value of the mask this can cause the buffer data to be written to the device.

Parameters **mask** (*constants.BufferOperation*) – Specifies the action to be taken with flushing the buffer. See highlevel.VisaLibraryBase.flush for a detailed description.

get_visa_attribute(*name: pyvisa.constants.ResourceAttribute*) → Any
Retrieves the state of an attribute in this resource.

One should prefer the dedicated descriptor for often used attributes since those perform checks and automatic conversion on the value.

Parameters **name** (*constants.ResourceAttribute*) – Resource attribute for which the state query is made.

Returns The state of the queried attribute for a specified resource.

Return type Any

ignore_warning (**warnings_constants*) → AbstractContextManager[T_co]
Ignoring warnings context manager for the current resource.

Parameters **warnings_constants** (constants.StatusCode) – Constants identifying the warnings to ignore.

implementation_version
Resource version that identifies the revisions or implementations of a resource.

This attribute value is defined by the individual manufacturer and increments with each new revision. The format of the value has the upper 12 bits as the major number of the version, the next lower 12 bits as the minor number of the version, and the lowest 8 bits as the sub-minor number of the version.

VISA Attribute VI_ATTR_RSRC_IMPL_VERSION (1073676291)

Type int

Range 0 <= value <= 4294967295

install_handler (*event_type: pyvisa.constants.EventType, handler: Callable[[NewType.<locals>.new_type, pyvisa.constants.EventType, NewType.<locals>.new_type, Any], None], user_handle=None*) → Any
Install handlers for event callbacks in this resource.

Parameters

- **event_type** (*constants.EventType*) – Logical event identifier.
- **handler** (*VISAHandler*) – Handler function to be installed by a client application.
- **user_handle** – A value specified by an application that can be used for identifying handlers uniquely for an event type. Depending on the backend they may be restriction on the possible values. Look at the backend *install_visa_handler* for more details.

Returns User handle in a format amenable to the backend. This is this representation of the handle that should be used when unistalling a handler.

Return type Any

interface_number
USB interface number used by the given session. :VISA Attribute: VI_ATTR_USB_INTFC_NUM (1073676705) :type: int :range: 0 <= value <= 254

interface_type
Interface type of the given session. :VISA Attribute: VI_ATTR_INTF_TYPE (1073676657) :type: :class:pyvisa.constants.InterfaceType

io_protocol
IO protocol to use.

In VXI, you can choose normal word serial or fast data channel (FDC). In GPIB, you can choose normal or high-speed (HS-488) transfers. In serial, TCPIP, or USB RAW, you can choose normal

transfers or 488.2-defined strings. In USB INSTR, you can choose normal or vendor-specific transfers.

VISA Attribute VI_ATTR_IO_PROT (1073676316)

Type :class:pyvisa.constants.IOProtocol

last_status
Last status code for this session.

lock(*timeout: Union[float, typing_extensions.Literal['default'][default]] = 'default', requested_key: Optional[str] = None*) → str
Establish a shared lock to the resource.

Parameters

- **timeout** (*Union[float, Literal["default"]], optional*) – Absolute time period (in milliseconds) that a resource waits to get unlocked by the locking session before returning an error. Defaults to "default" which means use self.timeout.
- **requested_key** (*Optional[str], optional*) – Access key used by another session with which you want your session to share a lock or None to generate a new shared access key.

Returns A new shared access key if requested_key is None, otherwise, same value as the requested_key

Return type str

lock_context(*timeout: Union[float, typing_extensions.Literal['default'][default]] = 'default', requested_key: Optional[str] = 'exclusive'*) → Iterator[Optional[str]]
A context that locks

Parameters

- **timeout** (*Union[float, Literal["default"]], optional*) – Absolute time period (in milliseconds) that a resource waits to get unlocked by the locking session before returning an error. Defaults to "default" which means use self.timeout.
- **requested_key** (*Optional[str], optional*) – When using default of 'exclusive' the lock is an exclusive lock. Otherwise it is the access key for the shared lock or None to generate a new shared access key.

Yields *Optional[str]* – The access_key if applicable.

lock_excl(*timeout: Union[float, typing_extensions.Literal['default'][default]] = 'default'*) → None
Establish an exclusive lock to the resource.

Parameters **timeout** (*Union[float, Literal["default"]], optional*) – Absolute time period (in milliseconds) that a resource waits to get unlocked by the locking session before returning an error. Defaults to "default" which means use self.timeout.

lock_state
Current locking state of the resource.

The resource can be unlocked, locked with an exclusive lock, or locked with a shared lock.

VISA Attribute VI_ATTR_RSRC_LOCK_STATE (1073676292)

Type :class:pyvisa.constants.AccessModes

manufacturer_id
Manufacturer identification number of the device. :VISA Attribute: VI_ATTR_MANF_ID (1073676505) :type: int :range: 0 <= value <= 65535

manufacturer_name
Manufacturer name. :VISA Attribute: VI_ATTR_MANF_NAME (3221160050)

maximum_interrupt_size
Maximum size of data that will be stored by any given USB interrupt.

If a USB interrupt contains more data than this size, the data in excess of this size will be lost.

VISA Attribute VI_ATTR_USB_MAX_INTR_SIZE (1073676719)

Type int

Range 0 <= value <= 65535

model_code
Model code for the device. :VISA Attribute: VI_ATTR_MODEL_CODE (1073676511) :type: int :range: 0 <= value <= 65535

model_name
Model name of the device. :VISA Attribute: VI_ATTR_MODEL_NAME (3221160055)

open(*access_mode: pyvisa.constants.AccessModes = <AccessModes.no_lock: 0>*, *open_timeout: int = 5000*) → None
Opens a session to the specified resource.

Parameters

- **access_mode** (*constants.AccessModes, optional*) – Specifies the mode by which the resource is to be accessed. Defaults to constants.AccessModes.no_lock.
- **open_timeout** (*int, optional*) – If the access_mode parameter requests a lock, then this parameter specifies the absolute time period (in milliseconds) that the resource waits to get unlocked before this operation returns an error. Defaults to 5000.

query(*message: str*, *delay: Optional[float] = None*) → str
A combination of write(message) and read()

Parameters

- **message** (*str*) – The message to send.
- **delay** (*Optional[float], optional*) – Delay in seconds between write and read operations. If None, defaults to self.query_delay.

Returns Answer from the device.

Return type str

query_ascii_values(*message: str, converter: Union[typing_extensions.Literal['s', 'b', 'c', 'd', 'o', 'x', 'X', 'e', 'E', 'f', 'F', 'g', 'G'][s, b, c, d, o, x, X, e, E, f, F, g, G], Callable[[str], Any]] = 'f', separator: Union[str, Callable[[str], Iterable[str]]] = ',', container: Union[Type[CT_co], Callable[[Iterable[T_co]], Sequence[T_co]]] = <class 'list'>, delay: Optional[float] = None*) → Sequence[Any]
Query the device for values in ascii format returning an iterable of values.

Parameters

- **message** (*str*) – The message to send.

- **converter** (*ASCII_CONVERTER, optional*) – Str format of function to convert each value. Default to "f".
- **separator** (*Union[str, Callable[[str], Iterable[str]]]*) – str or callable used to split the data into individual elements. If a str is given, data.split(separator) is used. Default to ",".
- **container** (*Union[Type, Callable[[Iterable], Sequence]], optional*) – Container type to use for the output data. Possible values are: list, tuple, np.ndarray, etc, Default to list.
- **delay** (*Optional[float], optional*) – Delay in seconds between write and read operations. If None, defaults to self.query_delay.

Returns Parsed data.

Return type Sequence

query_binary_values(*message: str, datatype: typing_extensions.Literal['s', 'b', 'B', 'h', 'H', 'i', 'I', 'l', 'L', 'q', 'Q', 'f', 'd'][s, b, B, h, H, i, I, l, L, q, Q, f, d] = 'f', is_big_endian: bool = False, container: Union[Type[CT_co], Callable[[Iterable[T_co]], Sequence[T_co]]] = <class 'list'>, delay: Optional[float] = None, header_fmt: typing_extensions.Literal['ieee', 'hp', 'empty'][ieee, hp, empty] = 'ieee', expect_termination: bool = True, data_points: int = 0, chunk_size: Optional[int] = None*) → Sequence[Union[int, float]]

Query the device for values in binary format returning an iterable of values.

Parameters

- **message** (*str*) – The message to send.
- **datatype** (*BINARY_DATATYPES, optional*) – Format string for a single element. See struct module. 'f' by default.
- **is_big_endian** (*bool, optional*) – Are the data in big or little endian order. Defaults to False.
- **container** (*Union[Type, Callable[[Iterable], Sequence]], optional*) – Container type to use for the output data. Possible values are: list, tuple, np.ndarray, etc, Default to list.
- **delay** (*Optional[float], optional*) – Delay in seconds between write and read operations. If None, defaults to self.query_delay.
- **header_fmt** (*util.BINARY_HEADERS, optional*) – Format of the header prefixing the data. Defaults to 'ieee'.
- **expect_termination** (*bool, optional*) – When set to False, the expected length of the binary values block does not account for the final termination character (the read termination). Defaults to True.
- **data_points** (*int, optional*) – Number of points expected in the block. This is used only if the instrument does not report it itself. This will be converted in a number of bytes based on the datatype. Defaults to 0.
- **chunk_size** (*int, optional*) – Size of the chunks to read from the device. Using larger chunks may be faster for large amount of data.

Returns Data read from the device.

Return type Sequence[Union[int, float]]

query_delay = 0.0

read(*termination: Optional[str] = None*, *encoding: Optional[str] = None*) → str
Read a string from the device.

Reading stops when the device stops sending (e.g. by setting appropriate bus lines), or the termination characters sequence was detected. Attention: Only the last character of the termination characters is really used to stop reading, however, the whole sequence is compared to the ending of the read string message. If they don't match, a warning is issued.

Parameters

- **termination** (*Optional[str], optional*) – Alternative character termination to use. If None, the value of write_termination is used. Defaults to None.
- **encoding** (*Optional[str], optional*) – Alternative encoding to use to turn bytes into str. If None, the value of encoding is used. Defaults to None.

Returns Message read from the instrument and decoded.

Return type str

read_ascii_values(*converter: Union[typing_extensions.Literal['s', 'b', 'c', 'd', 'o', 'x', 'X', 'e', 'E', 'f', 'F', 'g', 'G'][s, b, c, d, o, x, X, e, E, f, F, g, G], Callable[[str], Any]] = 'f', separator: Union[str, Callable[[str], Iterable[str]]] = ',', container: Union[Type[CT_co], Callable[[Iterable[T_co]], Sequence[T_co]]] = <class 'list'>*) → Sequence[T_co]
Read values from the device in ascii format returning an iterable of values.

Parameters **converter** (*ASCII_CONVERTER, optional*) – Str format of function to convert each value. Default to "f".

separator [Union[str, Callable[[str], Iterable[str]]]] str or callable used to split the data into individual elements. If a str is given, data.split(separator) is used. Default to ",".

container [Union[Type, Callable[[Iterable], Sequence]], optional] Container type to use for the output data. Possible values are: list, tuple, np.ndarray, etc, Default to list.

Returns Parsed data.

Return type Sequence

read_binary_values(*datatype: typing_extensions.Literal['s', 'b', 'B', 'h', 'H', 'i', 'I', 'l', 'L', 'q', 'Q', 'f', 'd'][s, b, B, h, H, i, I, l, L, q, Q, f, d] = 'f', is_big_endian: bool = False, container: Union[Type[CT_co], Callable[[Iterable[T_co]], Sequence[T_co]]] = <class 'list'>, header_fmt: typing_extensions.Literal['ieee', 'hp', 'empty'][ieee, hp, empty] = 'ieee', expect_termination: bool = True, data_points: int = 0, chunk_size: Optional[int] = None*) → Sequence[Union[int, float]]
Read values from the device in binary format returning an iterable of values.

Parameters

- **datatype** (*BINARY_DATATYPES, optional*) – Format string for a single element. See struct module. 'f' by default.
- **is_big_endian** (*bool, optional*) – Are the data in big or little endian order. Defaults to False.
- **container** (*Union[Type, Callable[[Iterable], Sequence]], optional*) – Container type to use for the output data. Possible values are: list, tuple, np.ndarray, etc, Default to list.

- **header_fmt** (*util.BINARY_HEADERS, optional*) – Format of the header prefixing the data. Defaults to 'ieee'.
- **expect_termination** (*bool, optional*) – When set to False, the expected length of the binary values block does not account for the final termination character (the read termination). Defaults to True.
- **data_points** (*int, optional*) – Number of points expected in the block. This is used only if the instrument does not report it itself. This will be converted in a number of bytes based on the datatype. Defaults to 0.
- **chunk_size** (*int, optional*) – Size of the chunks to read from the device. Using larger chunks may be faster for large amount of data.

Returns Data read from the device.

Return type Sequence[Union[int, float]]

read_bytes(*count: int*, *chunk_size: Optional[int] = None*, *break_on_termchar: bool = False*) → bytes

Read a certain number of bytes from the instrument.

Parameters

- **count** (*int*) – The number of bytes to read from the instrument.
- **chunk_size** (*Optional[int], optional*) – The chunk size to use to perform the reading. If count > chunk_size multiple low level operations will be performed. Defaults to None, meaning the resource wide set value is set.
- **break_on_termchar** (*bool, optional*) – Should the reading stop when a termination character is encountered or when the message ends. Defaults to False.

Returns Bytes read from the instrument.

Return type bytes

read_raw(*size: Optional[int] = None*) → bytes

Read the unmodified string sent from the instrument to the computer.

In contrast to read(), no termination characters are stripped.

Parameters **size** (*Optional[int], optional*) – The chunk size to use to perform the reading. Defaults to None, meaning the resource wide set value is set.

Returns Bytes read from the instrument.

Return type bytes

read_stb() → int

Service request status register.

read_termination

Read termination character.

read_termination_context(*new_termination: str*) → Iterator[T_co]

classmethod register(*interface_type: pyvisa.constants.InterfaceType*, *resource_class: str*) → Callable[[Type[T]], Type[T]]

Create a decorator to register a class.

The class is associated to an interface type, resource class pair.

Parameters

- **interface_type** (constants.InterfaceType) – Interface type for which to register a wrapper class.
- **resource_class** (*str*) – Resource class for which to register a wrapper class.

Returns Decorator registering the class. Raises TypeError if some VISA attributes are missing on the registered class.

Return type Callable[[Type[T]], Type[T]]

resource_class
Resource class as defined by the canonical resource name.

Possible values are: INSTR, INTFC, SOCKET, RAW...

VISA Attribute VI_ATTR_RSRC_CLASS (3221159937)

resource_info
Get the extended information of this resource.

resource_manufacturer_name
Manufacturer name of the vendor that implemented the VISA library.

This attribute is not related to the device manufacturer attributes.

Note The value of this attribute is for display purposes only and not for programmatic decisions, as the value can differ between VISA implementations and/or revisions.

VISA Attribute VI_ATTR_RSRC_MANF_NAME (3221160308)

resource_name
Unique identifier for a resource compliant with the address structure. :VISA Attribute: VI_ATTR_RSRC_NAME (3221159938)

send_end
Should END be asserted during the transfer of the last byte of the buffer. :VISA Attribute: VI_ATTR_SEND_END_EN (1073676310) :type: bool

serial_number
USB serial number of this device. :VISA Attribute: VI_ATTR_USB_SERIAL_NUM (3221160352)

session
Resource session handle.

Raises errors.InvalidSession – Raised if session is closed.

set_visa_attribute(*name: pyvisa.constants.ResourceAttribute*, *state: Any*) → pyvisa.constants.StatusCode
Set the state of an attribute.

One should prefer the dedicated descriptor for often used attributes since those perform checks and automatic conversion on the value.

Parameters

- **name** (*constants.ResourceAttribute*) – Attribute for which the state is to be modified.
- **state** (*Any*) – The state of the attribute to be set for the specified object.

Returns Return value of the library call.

Return type *constants.StatusCode*

spec_version
Version of the VISA specification to which the implementation is compliant.

The format of the value has the upper 12 bits as the major number of the version, the next lower 12 bits as the minor number of the version, and the lowest 8 bits as the sub-minor number of the version. The current VISA specification defines the value to be 00300000h.

VISA Attribute VI_ATTR_RSRC_SPEC_VERSION (1073676656)

Type int

Range 0 <= value <= 4294967295

stb
Service request status register.

timeout
Timeout in milliseconds for all resource I/O operations.

This value is used when accessing the device associated with the given session.

Special values:

- **immediate (`VI_TMO_IMMEDIATE`): 0** (for convenience, any value smaller than 1 is considered as 0)
- **infinite (`VI_TMO_INFINITE`): `float('+inf')`** (for convenience, None is considered as `float('+inf')`)

To set an **infinite** timeout, you can also use:

```
>>> del instrument.timeout
```

A timeout value of VI_TMO_IMMEDIATE means that operations should never wait for the device to respond. A timeout value of VI_TMO_INFINITE disables the timeout mechanism.

VISA Attribute VI_ATTR_TMO_VALUE (1073676314)

Type int

Range 0 <= value <= 4294967295

uninstall_handler(*event_type: pyvisa.constants.EventType, handler: Callable[[NewType.<locals>.new_type, pyvisa.constants.EventType, NewType.<locals>.new_type, Any], None], user_handle=None*) → None
Uninstalls handlers for events in this resource.

Parameters

- **event_type** (*constants.EventType*) – Logical event identifier.
- **handler** (*VISAHandler*) – Handler function to be uninstalled by a client application.
- **user_handle** (*Any*) – The user handle returned by install_handler.

unlock() → None
Relinquishes a lock for the specified resource.

usb_protocol
USB protocol used by this USB interface. :VISA Attribute: VI_ATTR_USB_PROTOCOL (1073676711) :type: int :range: 0 <= value <= 255

visa_attributes_classes = {<class 'pyvisa.attributes.AttrVI_ATTR_INTF_INST_NAME'>, <cl

wait_on_event(*in_event_type: pyvisa.constants.EventType*, *timeout: int*, *capture_timeout: bool = False*) → pyvisa.resources.resource.WaitResponse

Waits for an occurrence of the specified event in this resource.

in_event_type [constants.EventType] Logical identifier of the event(s) to wait for.

timeout [int] Absolute time period in time units that the resource shall wait for a specified event to occur before returning the time elapsed error. The time unit is in milliseconds. None means waiting forever if necessary.

capture_timeout [bool, optional] When True will not produce a VisaIOError(VI_ERROR_TMO) but instead return a WaitResponse with timed_out=True.

Returns Object that contains event_type, context and ret value.

Return type WaitResponse

wrap_handler(*callable: Callable[[Resource, pyvisa.events.Event, Any], None]*) → Callable[[NewType.<locals>.new_type, pyvisa.constants.EventType, NewType.<locals>.new_type, Any], None]

Wrap an event handler to provide the signature expected by VISA.

The handler is expected to have the following signature: handler(resource: Resource, event: Event, user_handle: Any) -> None.

The wrapped handler should be used only to handle events on the resource used to wrap the handler.

write(*message: str*, *termination: Optional[str] = None*, *encoding: Optional[str] = None*) → int

Write a string message to the device.

The write_termination is always appended to it.

Parameters

- **message** (*str*) – The message to be sent.
- **termination** (*Optional[str], optional*) – Alternative character termination to use. If None, the value of write_termination is used. Defaults to None.
- **encoding** (*Optional[str], optional*) – Alternative encoding to use to turn str into bytes. If None, the value of encoding is used. Defaults to None.

Returns Number of bytes written.

Return type int

write_ascii_values(*message: str, values: Sequence[Any], converter: Union[typing_extensions.Literal['s', 'b', 'c', 'd', 'o', 'x', 'X', 'e', 'E', 'f', 'F', 'g', 'G'][s, b, c, d, o, x, X, e, E, f, F, g, G], Callable[[str], Any]] = 'f', separator: Union[str, Callable[[Iterable[str]], str]] = ',', termination: Optional[str] = None, encoding: Optional[str] = None*)

Write a string message to the device followed by values in ascii format.

The write_termination is always appended to it.

Parameters

- **message** (*str*) – Header of the message to be sent.
- **values** (*Sequence[Any]*) – Data to be writen to the device.
- **converter** (*Union[str, Callable[[Any], str]], optional*) – Str formatting codes or function used to convert each value. Defaults to "f".

- **separator** (*Union[str, Callable[[Iterable[str]], str]], optional*) – Str or callable that join the values in a single str. If a str is given, separator.join(values) is used. Defaults to ','
- **termination** (*Optional[str], optional*) – Alternative character termination to use. If None, the value of write_termination is used. Defaults to None.
- **encoding** (*Optional[str], optional*) – Alternative encoding to use to turn str into bytes. If None, the value of encoding is used. Defaults to None.

Returns Number of bytes written.

Return type int

write_binary_values(*message: str, values: Sequence[Any], datatype: typing_extensions.Literal['s', 'b', 'B', 'h', 'H', 'i', 'I', 'l', 'L', 'q', 'Q', 'f', 'd'][s, b, B, h, H, i, I, l, L, q, Q, f, d] = 'f', is_big_endian: bool = False, termination: Optional[str] = None, encoding: Optional[str] = None, header_fmt: typing_extensions.Literal['ieee', 'hp', 'empty'][ieee, hp, empty] = 'ieee'*)

Write a string message to the device followed by values in binary format.

The write_termination is always appended to it.

Parameters

- **message** (*str*) – The header of the message to be sent.
- **values** (*Sequence[Any]*) – Data to be written to the device.
- **datatype** (*util.BINARY_DATATYPES, optional*) – The format string for a single element. See struct module.
- **is_big_endian** (*bool, optional*) – Are the data in big or little endian order.
- **termination** (*Optional[str], optional*) – Alternative character termination to use. If None, the value of write_termination is used. Defaults to None.
- **encoding** (*Optional[str], optional*) – Alternative encoding to use to turn str into bytes. If None, the value of encoding is used. Defaults to None.
- **header_fmt** (*util.BINARY_HEADERS*) – Format of the header prefixing the data.

Returns Number of bytes written.

Return type int

write_raw(*message: bytes*) → int

Write a byte message to the device.

Parameters **message** (*bytes*) – The message to be sent.

Returns Number of bytes written

Return type int

write_termination

Write termination character.

class pyvisa.resources.**GPIBInstrument**(*resource_manager: pyvisa.highlevel.ResourceManager, resource_name: str*)

Communicates with to devices of type GPIB::<primary address>[::INSTR]

More complex resource names can be specified with the following grammar: GPIB[board]::primary address[::secondary address][::INSTR]

Do not instantiate directly, use *pyvisa.highlevel.ResourceManager.open_resource()*.

CR = '\r'

LF = '\n'

allow_dma
Should I/O accesses use DMA (True) or Programmed I/O (False).

In some implementations, this attribute may have global effects even though it is documented to be a local attribute. Since this affects performance and not functionality, that behavior is acceptable.

VISA Attribute VI_ATTR_DMA_ALLOW_EN (1073676318)

Type bool

assert_trigger() → None
Sends a software trigger to the device.

before_close() → None
Called just before closing an instrument.

chunk_size = 20480

clear() → None
Clear this resource.

close() → None
Closes the VISA session and marks the handle as invalid.

control_atn(*mode: pyvisa.constants.ATNLineOperation*) → pyvisa.constants.StatusCode
Specifies the state of the ATN line and the local active controller state.

Corresponds to viGpibControlATN function of the VISA library.

Parameters **mode** (*constants.ATNLineOperation*) –

Specifies the state of the ATN line and optionally the local active controller state.

Returns Return value of the library call.

Return type *constants.StatusCode*

control_ren(*mode: pyvisa.constants.RENLineOperation*) → pyvisa.constants.StatusCode
Controls the state of the GPIB Remote Enable (REN) interface line.

The remote/local state of the device can also be controlled optionally.

Corresponds to viGpibControlREN function of the VISA library.

Parameters **mode** (*constants.RENLineOperation*) – Specifies the state of the REN line and optionally the device remote/local state.

Returns Return value of the library call.

Return type *constants.StatusCode*

disable_event(*event_type: pyvisa.constants.EventType*, *mechanism: pyvisa.constants.EventMechanism*) → None
Disable notification for an event type(s) via the specified mechanism(s).

Parameters

- **event_type** (*constants.EventType*) – Logical event identifier.

- **mechanism** (*constants.EventMechanism*) – Specifies event handling mechanisms to be disabled.

discard_events(*event_type: pyvisa.constants.EventType*, *mechanism: pyvisa.constants.EventMechanism*) → None
Discards event occurrences for an event type and mechanism in this resource.

Parameters

- **event_type** (*constants.EventType*) – Logical event identifier.
- **mechanism** (*constants.EventMechanism*) – Specifies event handling mechanisms to be disabled.

enable_event(*event_type: pyvisa.constants.EventType*, *mechanism: pyvisa.constants.EventMechanism*, *context: None = None*) → None
Enable event occurrences for specified event types and mechanisms in this resource.

Parameters

- **event_type** (*constants.EventType*) – Logical event identifier.
- **mechanism** (*constants.EventMechanism*) – Specifies event handling mechanisms to be enabled
- **context** (*None*) – Not currently used, leave as None.

enable_repeat_addressing
Whether to use repeat addressing before each read or write operation.

enable_unaddressing
Whether to unaddress the device (UNT and UNL) after each read or write operation.

encoding
Encoding used for read and write operations.

flush(*mask: pyvisa.constants.BufferOperation*) → None
Manually clears the specified buffers.

Depending on the value of the mask this can cause the buffer data to be written to the device.

Parameters **mask** (*constants.BufferOperation*) – Specifies the action to be taken with flushing the buffer. See highlevel.VisaLibraryBase.flush for a detailed description.

get_visa_attribute(*name: pyvisa.constants.ResourceAttribute*) → Any
Retrieves the state of an attribute in this resource.

One should prefer the dedicated descriptor for often used attributes since those perform checks and automatic conversion on the value.

Parameters **name** (*constants.ResourceAttribute*) – Resource attribute for which the state query is made.

Returns The state of the queried attribute for a specified resource.

Return type Any

ignore_warning(**warnings_constants*) → AbstractContextManager[T_co]
Ignoring warnings context manager for the current resource.

Parameters **warnings_constants** (*constants.StatusCode*) – Constants identifying the warnings to ignore.

implementation_version
Resource version that identifies the revisions or implementations of a resource.

This attribute value is defined by the individual manufacturer and increments with each new revision. The format of the value has the upper 12 bits as the major number of the version, the next lower 12 bits as the minor number of the version, and the lowest 8 bits as the sub-minor number of the version.

VISA Attribute VI_ATTR_RSRC_IMPL_VERSION (1073676291)

Type int

Range 0 <= value <= 4294967295

install_handler(*event_type: pyvisa.constants.EventType, handler: Callable[[NewType.<locals>.new_type, pyvisa.constants.EventType, NewType.<locals>.new_type, Any], None], user_handle=None*) → Any

Install handlers for event callbacks in this resource.

Parameters

- **event_type** (*constants.EventType*) – Logical event identifier.
- **handler** (*VISAHandler*) – Handler function to be installed by a client application.
- **user_handle** – A value specified by an application that can be used for identifying handlers uniquely for an event type. Depending on the backend they may be restriction on the possible values. Look at the backend *install_visa_handler* for more details.

Returns User handle in a format amenable to the backend. This is this representation of the handle that should be used when unistalling a handler.

Return type Any

interface_number

Board number for the given interface. :VISA Attribute: VI_ATTR_INTF_NUM (1073676662) :type: int :range: 0 <= value <= 65535

interface_type

Interface type of the given session. :VISA Attribute: VI_ATTR_INTF_TYPE (1073676657) :type: :class:pyvisa.constants.InterfaceType

io_protocol

IO protocol to use.

In VXI, you can choose normal word serial or fast data channel (FDC). In GPIB, you can choose normal or high-speed (HS-488) transfers. In serial, TCPIP, or USB RAW, you can choose normal transfers or 488.2-defined strings. In USB INSTR, you can choose normal or vendor-specific transfers.

VISA Attribute VI_ATTR_IO_PROT (1073676316)

Type :class:pyvisa.constants.IOProtocol

last_status

Last status code for this session.

lock(*timeout: Union[float, typing_extensions.Literal['default'][default]] = 'default', requested_key: Optional[str] = None*) → str

Establish a shared lock to the resource.

Parameters

- **timeout** (*Union[float, Literal["default"]], optional*) – Absolute time period (in milliseconds) that a resource waits to get unlocked by the locking session before returning an error. Defaults to "default" which means use self.timeout.
- **requested_key** (*Optional[str], optional*) – Access key used by another session with which you want your session to share a lock or None to generate a new shared access key.

Returns A new shared access key if requested_key is None, otherwise, same value as the requested_key

Return type str

lock_context(*timeout: Union[float, typing_extensions.Literal['default'][default]] = 'default', requested_key: Optional[str] = 'exclusive'*) → Iterator[Optional[str]]

A context that locks

Parameters

- **timeout** (*Union[float, Literal["default"]], optional*) – Absolute time period (in milliseconds) that a resource waits to get unlocked by the locking session before returning an error. Defaults to "default" which means use self.timeout.
- **requested_key** (*Optional[str], optional*) – When using default of 'exclusive' the lock is an exclusive lock. Otherwise it is the access key for the shared lock or None to generate a new shared access key.

Yields *Optional[str]* – The access_key if applicable.

lock_excl(*timeout: Union[float, typing_extensions.Literal['default'][default]] = 'default'*) → None

Establish an exclusive lock to the resource.

Parameters **timeout** (*Union[float, Literal["default"]], optional*) – Absolute time period (in milliseconds) that a resource waits to get unlocked by the locking session before returning an error. Defaults to "default" which means use self.timeout.

lock_state

Current locking state of the resource.

The resource can be unlocked, locked with an exclusive lock, or locked with a shared lock.

VISA Attribute VI_ATTR_RSRC_LOCK_STATE (1073676292)

Type :class:pyvisa.constants.AccessModes

open(*access_mode: pyvisa.constants.AccessModes = <AccessModes.no_lock: 0>, open_timeout: int = 5000*) → None

Opens a session to the specified resource.

Parameters

- **access_mode** (*constants.AccessModes, optional*) – Specifies the mode by which the resource is to be accessed. Defaults to constants.AccessModes.no_lock.
- **open_timeout** (*int, optional*) – If the access_mode parameter requests a lock, then this parameter specifies the absolute time period (in milliseconds) that the resource waits to get unlocked before this operation returns an error. Defaults to 5000.

pass_control(*primary_address: int, secondary_address: int*) → pyvisa.constants.StatusCode

Tell a GPIB device to become controller in charge (CIC).

Corresponds to viGpibPassControl function of the VISA library.

Parameters

- **primary_address** (*int*) – Primary address of the GPIB device to which you want to pass control.
- **secondary_address** (*int*) – Secondary address of the targeted GPIB device. If the targeted device does not have a secondary address, this parameter should contain the value Constants.NO_SEC_ADDR.

Returns Return value of the library call.

Return type *constants.StatusCode*

primary_address
Primary address of the GPIB device used by the given session.

For the GPIB INTFC Resource, this attribute is Read-Write.

VISA Attribute VI_ATTR_GPIB_PRIMARY_ADDR (1073676658)

Type int

Range 0 <= value <= 30

query(*message: str*, *delay: Optional[float] = None*) → str
A combination of write(message) and read()

Parameters

- **message** (*str*) – The message to send.
- **delay** (*Optional[float], optional*) – Delay in seconds between write and read operations. If None, defaults to self.query_delay.

Returns Answer from the device.

Return type str

query_ascii_values(*message: str, converter: Union[typing_extensions.Literal['s', 'b', 'c', 'd', 'o', 'x', 'X', 'e', 'E', 'f', 'F', 'g', 'G'][s, b, c, d, o, x, X, e, E, f, F, g, G], Callable[[str], Any]] = 'f', separator: Union[str, Callable[[str], Iterable[str]]] = ',', container: Union[Type[CT_co], Callable[[Iterable[T_co]], Sequence[T_co]]] = <class 'list'>, delay: Optional[float] = None*) → Sequence[Any]
Query the device for values in ascii format returning an iterable of values.

Parameters

- **message** (*str*) – The message to send.
- **converter** (*ASCII_CONVERTER, optional*) – Str format of function to convert each value. Default to "f".
- **separator** (*Union[str, Callable[[str], Iterable[str]]]*) – str or callable used to split the data into individual elements. If a str is given, data.split(separator) is used. Default to ",".
- **container** (*Union[Type, Callable[[Iterable], Sequence]], optional*) – Container type to use for the output data. Possible values are: list, tuple, np.ndarray, etc, Default to list.
- **delay** (*Optional[float], optional*) – Delay in seconds between write and read operations. If None, defaults to self.query_delay.

Returns Parsed data.

Return type Sequence

query_binary_values(*message: str, datatype: typing_extensions.Literal['s', 'b', 'B', 'h', 'H', 'i', 'I', 'l', 'L', 'q', 'Q', 'f', 'd'][s, b, B, h, H, i, I, l, L, q, Q, f, d] = 'f', is_big_endian: bool = False, container: Union[Type[CT_co], Callable[[Iterable[T_co]], Sequence[T_co]]] = <class 'list'>, delay: Optional[float] = None, header_fmt: typing_extensions.Literal['ieee', 'hp', 'empty'][ieee, hp, empty] = 'ieee', expect_termination: bool = True, data_points: int = 0, chunk_size: Optional[int] = None*) → Sequence[Union[int, float]]

Query the device for values in binary format returning an iterable of values.

Parameters

- **message** (*str*) – The message to send.
- **datatype** (*BINARY_DATATYPES, optional*) – Format string for a single element. See struct module. 'f' by default.
- **is_big_endian** (*bool, optional*) – Are the data in big or little endian order. Defaults to False.
- **container** (*Union[Type, Callable[[Iterable], Sequence]], optional*) – Container type to use for the output data. Possible values are: list, tuple, np.ndarray, etc, Default to list.
- **delay** (*Optional[float], optional*) – Delay in seconds between write and read operations. If None, defaults to self.query_delay.
- **header_fmt** (*util.BINARY_HEADERS, optional*) – Format of the header prefixing the data. Defaults to 'ieee'.
- **expect_termination** (*bool, optional*) – When set to False, the expected length of the binary values block does not account for the final termination character (the read termination). Defaults to True.
- **data_points** (*int, optional*) – Number of points expected in the block. This is used only if the instrument does not report it itself. This will be converted in a number of bytes based on the datatype. Defaults to 0.
- **chunk_size** (*int, optional*) – Size of the chunks to read from the device. Using larger chunks may be faster for large amount of data.

Returns Data read from the device.

Return type Sequence[Union[int, float]]

query_delay = 0.0

read(*termination: Optional[str] = None, encoding: Optional[str] = None*) → str

Read a string from the device.

Reading stops when the device stops sending (e.g. by setting appropriate bus lines), or the termination characters sequence was detected. Attention: Only the last character of the termination characters is really used to stop reading, however, the whole sequence is compared to the ending of the read string message. If they don't match, a warning is issued.

Parameters

- **termination** (*Optional[str], optional*) – Alternative character termination to use. If None, the value of write_termination is used. Defaults to None.
- **encoding** (*Optional[str], optional*) – Alternative encoding to use to turn bytes into str. If None, the value of encoding is used. Defaults to None.

Returns Message read from the instrument and decoded.

Return type str

read_ascii_values(*converter: Union[typing_extensions.Literal['s', 'b', 'c', 'd', 'o', 'x', 'X', 'e', 'E', 'f', 'F', 'g', 'G']][s, b, c, d, o, x, X, e, E, f, F, g, G], Callable[[str], Any]] = 'f', separator: Union[str, Callable[[str], Iterable[str]]] = ',', container: Union[Type[CT_co], Callable[[Iterable[T_co]], Sequence[T_co]]] = <class 'list'>*) → Sequence[T_co]

Read values from the device in ascii format returning an iterable of values.

Parameters **converter** (*ASCII_CONVERTER, optional*) – Str format of function to convert each value. Default to "f".

separator [Union[str, Callable[[str], Iterable[str]]]] str or callable used to split the data into individual elements. If a str is given, data.split(separator) is used. Default to ",".

container [Union[Type, Callable[[Iterable], Sequence]], optional] Container type to use for the output data. Possible values are: list, tuple, np.ndarray, etc, Default to list.

Returns Parsed data.

Return type Sequence

read_binary_values(*datatype: typing_extensions.Literal['s', 'b', 'B', 'h', 'H', 'i', 'I', 'l', 'L', 'q', 'Q', 'f', 'd'][s, b, B, h, H, i, I, l, L, q, Q, f, d] = 'f', is_big_endian: bool = False, container: Union[Type[CT_co], Callable[[Iterable[T_co]], Sequence[T_co]]] = <class 'list'>, header_fmt: typing_extensions.Literal['ieee', 'hp', 'empty'][ieee, hp, empty] = 'ieee', expect_termination: bool = True, data_points: int = 0, chunk_size: Optional[int] = None*) → Sequence[Union[int, float]]

Read values from the device in binary format returning an iterable of values.

Parameters

- **datatype** (*BINARY_DATATYPES, optional*) – Format string for a single element. See struct module. 'f' by default.
- **is_big_endian** (*bool, optional*) – Are the data in big or little endian order. Defaults to False.
- **container** (*Union[Type, Callable[[Iterable], Sequence]], optional*) – Container type to use for the output data. Possible values are: list, tuple, np.ndarray, etc, Default to list.
- **header_fmt** (*util.BINARY_HEADERS, optional*) – Format of the header prefixing the data. Defaults to 'ieee'.
- **expect_termination** (*bool, optional*) – When set to False, the expected length of the binary values block does not account for the final termination character (the read termination). Defaults to True.
- **data_points** (*int, optional*) – Number of points expected in the block. This is used only if the instrument does not report it itself. This will be converted in a number of bytes based on the datatype. Defaults to 0.
- **chunk_size** (*int, optional*) – Size of the chunks to read from the device. Using larger chunks may be faster for large amount of data.

Returns Data read from the device.

Return type Sequence[Union[int, float]]

read_bytes(*count: int*, *chunk_size: Optional[int] = None*, *break_on_termchar: bool = False*) → bytes

Read a certain number of bytes from the instrument.

Parameters

- **count** (*int*) – The number of bytes to read from the instrument.
- **chunk_size** (*Optional[int], optional*) – The chunk size to use to perform the reading. If count > chunk_size multiple low level operations will be performed. Defaults to None, meaning the resource wide set value is set.
- **break_on_termchar** (*bool, optional*) – Should the reading stop when a termination character is encountered or when the message ends. Defaults to False.

Returns Bytes read from the instrument.

Return type bytes

read_raw(*size: Optional[int] = None*) → bytes

Read the unmodified string sent from the instrument to the computer.

In contrast to read(), no termination characters are stripped.

Parameters **size** (*Optional[int], optional*) – The chunk size to use to perform the reading. Defaults to None, meaning the resource wide set value is set.

Returns Bytes read from the instrument.

Return type bytes

read_stb() → int

Service request status register.

read_termination

Read termination character.

read_termination_context(*new_termination: str*) → Iterator[T_co]

classmethod register(*interface_type: pyvisa.constants.InterfaceType*, *resource_class: str*) → Callable[[Type[T]], Type[T]]

Create a decorator to register a class.

The class is associated to an interface type, resource class pair.

Parameters

- **interface_type** (constants.InterfaceType) – Interface type for which to register a wrapper class.
- **resource_class** (*str*) – Resource class for which to register a wrapper class.

Returns Decorator registering the class. Raises TypeError if some VISA attributes are missing on the registered class.

Return type Callable[[Type[T]], Type[T]]

remote_enabled

Current state of the GPIB REN (Remote ENable) interface line. :VISA Attribute: VI_ATTR_GPIB_REN_STATE (1073676673) :type: :class:pyvisa.constants.LineState

resource_class

Resource class as defined by the canonical resource name.

Possible values are: INSTR, INTFC, SOCKET, RAW...

VISA Attribute VI_ATTR_RSRC_CLASS (3221159937)

resource_info
Get the extended information of this resource.

resource_manufacturer_name
Manufacturer name of the vendor that implemented the VISA library.

This attribute is not related to the device manufacturer attributes.

Note The value of this attribute is for display purposes only and not for programmatic decisions, as the value can differ between VISA implementations and/or revisions.

VISA Attribute VI_ATTR_RSRC_MANF_NAME (3221160308)

resource_name
Unique identifier for a resource compliant with the address structure. :VISA Attribute: VI_ATTR_RSRC_NAME (3221159938)

secondary_address
Secondary address of the GPIB device used by the given session.

For the GPIB INTFC Resource, this attribute is Read-Write.

VISA Attribute VI_ATTR_GPIB_SECONDARY_ADDR (1073676659)

Type int

Range 0 <= value <= 30 or in [65535]

send_command(*data: bytes*) → Tuple[int, pyvisa.constants.StatusCode]
Write GPIB command bytes on the bus.

Corresponds to viGpibCommand function of the VISA library.

Parameters **data** (*bytes*) – Command to write.

Returns

- *int* – Number of bytes written
- *constants.StatusCode* – Return value of the library call.

send_end
Should END be asserted during the transfer of the last byte of the buffer. :VISA Attribute: VI_ATTR_SEND_END_EN (1073676310) :type: bool

send_ifc() → pyvisa.constants.StatusCode
Pulse the interface clear line (IFC) for at least 100 microseconds.

Corresponds to viGpibSendIFC function of the VISA library.

session
Resource session handle.

Raises errors.InvalidSession – Raised if session is closed.

set_visa_attribute(*name: pyvisa.constants.ResourceAttribute*, *state: Any*) → pyvisa.constants.StatusCode
Set the state of an attribute.

One should prefer the dedicated descriptor for often used attributes since those perform checks and automatic conversion on the value.

Parameters

- **name** (*constants.ResourceAttribute*) – Attribute for which the state is to be modified.
- **state** (*Any*) – The state of the attribute to be set for the specified object.

Returns Return value of the library call.

Return type *constants.StatusCode*

spec_version

Version of the VISA specification to which the implementation is compliant.

The format of the value has the upper 12 bits as the major number of the version, the next lower 12 bits as the minor number of the version, and the lowest 8 bits as the sub-minor number of the version. The current VISA specification defines the value to be 00300000h.

VISA Attribute VI_ATTR_RSRC_SPEC_VERSION (1073676656)

Type int

Range 0 <= value <= 4294967295

stb

Service request status register.

timeout

Timeout in milliseconds for all resource I/O operations.

This value is used when accessing the device associated with the given session.

Special values:

- **immediate (VI_TMO_IMMEDIATE): 0** (for convenience, any value smaller than 1 is considered as 0)
- **infinite (VI_TMO_INFINITE): float('+inf')** (for convenience, None is considered as `float('+inf')`)

To set an **infinite** timeout, you can also use:

```
>>> del instrument.timeout
```

A timeout value of VI_TMO_IMMEDIATE means that operations should never wait for the device to respond. A timeout value of VI_TMO_INFINITE disables the timeout mechanism.

VISA Attribute VI_ATTR_TMO_VALUE (1073676314)

Type int

Range 0 <= value <= 4294967295

uninstall_handler(*event_type: pyvisa.constants.EventType, handler: Callable[[NewType.<locals>.new_type, pyvisa.constants.EventType, NewType.<locals>.new_type, Any], None], user_handle=None*) → None

Uninstalls handlers for events in this resource.

Parameters

- **event_type** (*constants.EventType*) – Logical event identifier.
- **handler** (*VISAHandler*) – Handler function to be uninstalled by a client application.

- **user_handle** (*Any*) – The user handle returned by install_handler.

unlock() → None
Relinquishes a lock for the specified resource.

visa_attributes_classes = {<class 'pyvisa.attributes.AttrVI_ATTR_INTF_INST_NAME'>, <cl

wait_for_srq(*timeout: int = 25000*) → None
Wait for a serial request (SRQ) coming from the instrument.

Note that this method is not ended when *another* instrument signals an SRQ, only *this* instrument.

Parameters **timeout** (*int*) – Maximum waiting time in milliseconds. Defaul: 25000 (milliseconds). None means waiting forever if necessary.

wait_on_event(*in_event_type: pyvisa.constants.EventType*, *timeout: int*, *capture_timeout: bool = False*) → pyvisa.resources.resource.WaitResponse
Waits for an occurrence of the specified event in this resource.

in_event_type [constants.EventType] Logical identifier of the event(s) to wait for.

timeout [int] Absolute time period in time units that the resource shall wait for a specified event to occur before returning the time elapsed error. The time unit is in milliseconds. None means waiting forever if necessary.

capture_timeout [bool, optional] When True will not produce a VisaIOError(VI_ERROR_TMO) but instead return a WaitResponse with timed_out=True.

Returns Object that contains event_type, context and ret value.

Return type WaitResponse

wrap_handler(*callable: Callable[[Resource, pyvisa.events.Event, Any], None]*) → Callable[[NewType.<locals>.new_type, pyvisa.constants.EventType, NewType.<locals>.new_type, Any], None]
Wrap an event handler to provide the signature expected by VISA.

The handler is expected to have the following signature: handler(resource: Resource, event: Event, user_handle: Any) -> None.

The wrapped handler should be used only to handle events on the resource used to wrap the handler.

write(*message: str*, *termination: Optional[str] = None*, *encoding: Optional[str] = None*) → int
Write a string message to the device.

The write_termination is always appended to it.

Parameters

- **message** (*str*) – The message to be sent.
- **termination** (*Optional[str], optional*) – Alternative character termination to use. If None, the value of write_termination is used. Defaults to None.
- **encoding** (*Optional[str], optional*) – Alternative encoding to use to turn str into bytes. If None, the value of encoding is used. Defaults to None.

Returns Number of bytes written.

Return type int

write_ascii_values(*message: str, values: Sequence[Any], converter: Union[typing_extensions.Literal['s', 'b', 'c', 'd', 'o', 'x', 'X', 'e', 'E', 'f', 'F', 'g', 'G'][s, b, c, d, o, x, X, e, E, f, F, g, G], Callable[[str], Any]] = 'f', separator: Union[str, Callable[[Iterable[str]], str]] = ',', termination: Optional[str] = None, encoding: Optional[str] = None*)

Write a string message to the device followed by values in ascii format.

The write_termination is always appended to it.

Parameters

- **message** (*str*) – Header of the message to be sent.
- **values** (*Sequence[Any]*) – Data to be writen to the device.
- **converter** (*Union[str, Callable[[Any], str]], optional*) – Str formatting codes or function used to convert each value. Defaults to "f".
- **separator** (*Union[str, Callable[[Iterable[str]], str]], optional*) – Str or callable that join the values in a single str. If a str is given, separator.join(values) is used. Defaults to ','
- **termination** (*Optional[str], optional*) – Alternative character termination to use. If None, the value of write_termination is used. Defaults to None.
- **encoding** (*Optional[str], optional*) – Alternative encoding to use to turn str into bytes. If None, the value of encoding is used. Defaults to None.

Returns Number of bytes written.

Return type int

write_binary_values(*message: str, values: Sequence[Any], datatype: typing_extensions.Literal['s', 'b', 'B', 'h', 'H', 'i', 'I', 'l', 'L', 'q', 'Q', 'f', 'd'][s, b, B, h, H, i, I, l, L, q, Q, f, d] = 'f', is_big_endian: bool = False, termination: Optional[str] = None, encoding: Optional[str] = None, header_fmt: typing_extensions.Literal['ieee', 'hp', 'empty'][ieee, hp, empty] = 'ieee'*)

Write a string message to the device followed by values in binary format.

The write_termination is always appended to it.

Parameters

- **message** (*str*) – The header of the message to be sent.
- **values** (*Sequence[Any]*) – Data to be written to the device.
- **datatype** (*util.BINARY_DATATYPES, optional*) – The format string for a single element. See struct module.
- **is_big_endian** (*bool, optional*) – Are the data in big or little endian order.
- **termination** (*Optional[str], optional*) – Alternative character termination to use. If None, the value of write_termination is used. Defaults to None.
- **encoding** (*Optional[str], optional*) – Alternative encoding to use to turn str into bytes. If None, the value of encoding is used. Defaults to None.
- **header_fmt** (*util.BINARY_HEADERS*) – Format of the header prefixing the data.

Returns Number of bytes written.

Return type int

write_raw(*message: bytes*) → int
Write a byte message to the device.

Parameters **message** (*bytes*) – The message to be sent.

Returns Number of bytes written

Return type int

write_termination
Write termination character.

class pyvisa.resources.**GPIBInterface**(*resource_manager: pyvisa.highlevel.ResourceManager*, *resource_name: str*)
Communicates with to devices of type GPIB::INTFC

More complex resource names can be specified with the following grammar: GPIB[board]::INTFC

Do not instantiate directly, use *pyvisa.highlevel.ResourceManager.open_resource()*.

CR = '\r'

LF = '\n'

address_state
Is the GPIB interface currently addressed to talk or listen, or is not addressed.

allow_dma
Should I/O accesses use DMA (True) or Programmed I/O (False).

In some implementations, this attribute may have global effects even though it is documented to be a local attribute. Since this affects performance and not functionality, that behavior is acceptable.

VISA Attribute VI_ATTR_DMA_ALLOW_EN (1073676318)

Type bool

assert_trigger() → None
Sends a software trigger to the device.

atn_state
Current state of the GPIB ATN (ATtentioN) interface line.

before_close() → None
Called just before closing an instrument.

chunk_size = 20480

clear() → None
Clear this resource.

close() → None
Closes the VISA session and marks the handle as invalid.

control_atn(*mode: pyvisa.constants.ATNLineOperation*) → pyvisa.constants.StatusCode
Specifies the state of the ATN line and the local active controller state.

Corresponds to viGpibControlATN function of the VISA library.

Parameters **mode** (*constants.ATNLineOperation*) –

Specifies the state of the ATN line and optionally the local active controller state.

Returns Return value of the library call.

Return type *constants.StatusCode*

control_ren(*mode: pyvisa.constants.RENLineOperation*) → pyvisa.constants.StatusCode
Controls the state of the GPIB Remote Enable (REN) interface line.

The remote/local state of the device can also be controlled optionally.

Corresponds to viGpibControlREN function of the VISA library.

Parameters **mode** (*constants.RENLineOperation*) – Specifies the state of the REN line and optionally the device remote/local state.

Returns Return value of the library call.

Return type *constants.StatusCode*

disable_event(*event_type: pyvisa.constants.EventType*, *mechanism: pyvisa.constants.EventMechanism*) → None
Disable notification for an event type(s) via the specified mechanism(s).

Parameters

- **event_type** (*constants.EventType*) – Logical event identifier.
- **mechanism** (*constants.EventMechanism*) – Specifies event handling mechanisms to be disabled.

discard_events(*event_type: pyvisa.constants.EventType*, *mechanism: pyvisa.constants.EventMechanism*) → None
Discards event occurrences for an event type and mechanism in this resource.

Parameters

- **event_type** (*constants.EventType*) – Logical event identifier.
- **mechanism** (*constants.EventMechanism*) – Specifies event handling mechanisms to be disabled.

enable_event(*event_type: pyvisa.constants.EventType*, *mechanism: pyvisa.constants.EventMechanism*, *context: None = None*) → None
Enable event occurrences for specified event types and mechanisms in this resource.

Parameters

- **event_type** (*constants.EventType*) – Logical event identifier.
- **mechanism** (*constants.EventMechanism*) – Specifies event handling mechanisms to be enabled
- **context** (*None*) – Not currently used, leave as None.

encoding
Encoding used for read and write operations.

flush(*mask: pyvisa.constants.BufferOperation*) → None
Manually clears the specified buffers.

Depending on the value of the mask this can cause the buffer data to be written to the device.

Parameters **mask** (*constants.BufferOperation*) – Specifies the action to be taken with flushing the buffer. See highlevel.VisaLibraryBase.flush for a detailed description.

get_visa_attribute(*name: pyvisa.constants.ResourceAttribute*) → Any
Retrieves the state of an attribute in this resource.

One should prefer the dedicated descriptor for often used attributes since those perform checks and automatic conversion on the value.

Parameters **name** (*constants.ResourceAttribute*) – Resource attribute for which the state query is made.

Returns The state of the queried attribute for a specified resource.

Return type Any

group_execute_trigger(**resources*) → Tuple[int, pyvisa.constants.StatusCode]

Parameters **resources** (GPIBInstrument) – GPIB resources to which to send the group trigger.

Returns

- *int* – Number of bytes written as part of sending the GPIB commands.
- *constants.StatusCode* – Return value of the library call.

ignore_warning(**warnings_constants*) → AbstractContextManager[T_co]

Ignoring warnings context manager for the current resource.

Parameters **warnings_constants** (constants.StatusCode) – Constants identifying the warnings to ignore.

implementation_version

Resource version that identifies the revisions or implementations of a resource.

This attribute value is defined by the individual manufacturer and increments with each new revision. The format of the value has the upper 12 bits as the major number of the version, the next lower 12 bits as the minor number of the version, and the lowest 8 bits as the sub-minor number of the version.

VISA Attribute VI_ATTR_RSRC_IMPL_VERSION (1073676291)

Type int

Range 0 <= value <= 4294967295

install_handler(*event_type: pyvisa.constants.EventType, handler: Callable[[NewType.<locals>.new_type, pyvisa.constants.EventType, NewType.<locals>.new_type, Any], None], user_handle=None*) → Any

Install handlers for event callbacks in this resource.

Parameters

- **event_type** (*constants.EventType*) – Logical event identifier.
- **handler** (*VISAHandler*) – Handler function to be installed by a client application.
- **user_handle** – A value specified by an application that can be used for identifying handlers uniquely for an event type. Depending on the backend they may be restriction on the possible values. Look at the backend *install_visa_handler* for more details.

Returns User handle in a format amenable to the backend. This is this representation of the handle that should be used when unistalling a handler.

Return type Any

interface_number

Board number for the given interface. :VISA Attribute: VI_ATTR_INTF_NUM (1073676662) :type: int :range: 0 <= value <= 65535

interface_type

Interface type of the given session. :VISA Attribute: VI_ATTR_INTF_TYPE (1073676657) :type: :class:pyvisa.constants.InterfaceType

io_protocol
IO protocol to use.

In VXI, you can choose normal word serial or fast data channel (FDC). In GPIB, you can choose normal or high-speed (HS-488) transfers. In serial, TCPIP, or USB RAW, you can choose normal transfers or 488.2-defined strings. In USB INSTR, you can choose normal or vendor-specific transfers.

VISA Attribute VI_ATTR_IO_PROT (1073676316)

Type :class:pyvisa.constants.IOProtocol

is_controller_in_charge
Is the specified GPIB interface currently CIC (Controller In Charge).

is_system_controller
Is the specified GPIB interface currently the system controller.

last_status
Last status code for this session.

lock(*timeout: Union[float, typing_extensions.Literal['default']][default]] = 'default', requested_key: Optional[str] = None*) → str
Establish a shared lock to the resource.

Parameters

- **timeout** (*Union[float, Literal["default"]], optional*) – Absolute time period (in milliseconds) that a resource waits to get unlocked by the locking session before returning an error. Defaults to "default" which means use self.timeout.
- **requested_key** (*Optional[str], optional*) – Access key used by another session with which you want your session to share a lock or None to generate a new shared access key.

Returns A new shared access key if requested_key is None, otherwise, same value as the requested_key

Return type str

lock_context(*timeout: Union[float, typing_extensions.Literal['default']][default]] = 'default', requested_key: Optional[str] = 'exclusive'*) → Iterator[Optional[str]]
A context that locks

Parameters

- **timeout** (*Union[float, Literal["default"]], optional*) – Absolute time period (in milliseconds) that a resource waits to get unlocked by the locking session before returning an error. Defaults to "default" which means use self.timeout.
- **requested_key** (*Optional[str], optional*) – When using default of 'exclusive' the lock is an exclusive lock. Otherwise it is the access key for the shared lock or None to generate a new shared access key.

Yields *Optional[str]* – The access_key if applicable.

lock_excl(*timeout: Union[float, typing_extensions.Literal['default']][default]] = 'default'*) → None
Establish an exclusive lock to the resource.

Parameters **timeout** (*Union[float, Literal["default"]], optional*) – Absolute time period (in milliseconds) that a resource waits to get unlocked by the locking session before returning an error. Defaults to "default" which means use self.timeout.

lock_state
Current locking state of the resource.

The resource can be unlocked, locked with an exclusive lock, or locked with a shared lock.

VISA Attribute VI_ATTR_RSRC_LOCK_STATE (1073676292)

Type :class:pyvisa.constants.AccessModes

ndac_state
Current state of the GPIB NDAC (Not Data ACcepted) interface line.

open(*access_mode: pyvisa.constants.AccessModes = <AccessModes.no_lock: 0>*, *open_timeout: int = 5000*) → None
Opens a session to the specified resource.

Parameters

- **access_mode** (*constants.AccessModes, optional*) – Specifies the mode by which the resource is to be accessed. Defaults to constants.AccessModes.no_lock.
- **open_timeout** (*int, optional*) – If the access_mode parameter requests a lock, then this parameter specifies the absolute time period (in milliseconds) that the resource waits to get unlocked before this operation returns an error. Defaults to 5000.

pass_control(*primary_address: int*, *secondary_address: int*) → pyvisa.constants.StatusCode
Tell a GPIB device to become controller in charge (CIC).

Corresponds to viGpibPassControl function of the VISA library.

Parameters

- **primary_address** (*int*) – Primary address of the GPIB device to which you want to pass control.
- **secondary_address** (*int*) – Secondary address of the targeted GPIB device. If the targeted device does not have a secondary address, this parameter should contain the value Constants.NO_SEC_ADDR.

Returns Return value of the library call.

Return type *constants.StatusCode*

primary_address
Primary address of the GPIB device used by the given session.

For the GPIB INTFC Resource, this attribute is Read-Write.

VISA Attribute VI_ATTR_GPIB_PRIMARY_ADDR (1073676658)

Type int

Range 0 <= value <= 30

query(*message: str*, *delay: Optional[float] = None*) → str
A combination of write(message) and read()

Parameters

- **message** (*str*) – The message to send.
- **delay** (*Optional[float], optional*) – Delay in seconds between write and read operations. If None, defaults to self.query_delay.

Returns Answer from the device.

Return type str

query_ascii_values(*message: str, converter: Union[typing_extensions.Literal['s', 'b', 'c', 'd', 'o', 'x', 'X', 'e', 'E', 'f', 'F', 'g', 'G'][s, b, c, d, o, x, X, e, E, f, F, g, G], Callable[[str], Any]] = 'f', separator: Union[str, Callable[[str], Iterable[str]]] = ',', container: Union[Type[CT_co], Callable[[Iterable[T_co]], Sequence[T_co]]] = <class 'list'>, delay: Optional[float] = None*) → Sequence[Any]

Query the device for values in ascii format returning an iterable of values.

Parameters

- **message** (*str*) – The message to send.
- **converter** (*ASCII_CONVERTER, optional*) – Str format of function to convert each value. Default to "f".
- **separator** (*Union[str, Callable[[str], Iterable[str]]]*) – str or callable used to split the data into individual elements. If a str is given, data.split(separator) is used. Default to ",".
- **container** (*Union[Type, Callable[[Iterable], Sequence]], optional*) – Container type to use for the output data. Possible values are: list, tuple, np.ndarray, etc, Default to list.
- **delay** (*Optional[float], optional*) – Delay in seconds between write and read operations. If None, defaults to self.query_delay.

Returns Parsed data.

Return type Sequence

query_binary_values(*message: str, datatype: typing_extensions.Literal['s', 'b', 'B', 'h', 'H', 'i', 'I', 'l', 'L', 'q', 'Q', 'f', 'd'][s, b, B, h, H, i, I, l, L, q, Q, f, d] = 'f', is_big_endian: bool = False, container: Union[Type[CT_co], Callable[[Iterable[T_co]], Sequence[T_co]]] = <class 'list'>, delay: Optional[float] = None, header_fmt: typing_extensions.Literal['ieee', 'hp', 'empty'][ieee, hp, empty] = 'ieee', expect_termination: bool = True, data_points: int = 0, chunk_size: Optional[int] = None*) → Sequence[Union[int, float]]

Query the device for values in binary format returning an iterable of values.

Parameters

- **message** (*str*) – The message to send.
- **datatype** (*BINARY_DATATYPES, optional*) – Format string for a single element. See struct module. 'f' by default.
- **is_big_endian** (*bool, optional*) – Are the data in big or little endian order. Defaults to False.
- **container** (*Union[Type, Callable[[Iterable], Sequence]], optional*) – Container type to use for the output data. Possible values are: list, tuple, np.ndarray, etc, Default to list.
- **delay** (*Optional[float], optional*) – Delay in seconds between write and read operations. If None, defaults to self.query_delay.
- **header_fmt** (*util.BINARY_HEADERS, optional*) – Format of the header prefixing the data. Defaults to 'ieee'.

- **expect_termination** (*bool, optional*) – When set to False, the expected length of the binary values block does not account for the final termination character (the read termination). Defaults to True.
- **data_points** (*int, optional*) – Number of points expected in the block. This is used only if the instrument does not report it itself. This will be converted in a number of bytes based on the datatype. Defaults to 0.
- **chunk_size** (*int, optional*) – Size of the chunks to read from the device. Using larger chunks may be faster for large amount of data.

Returns Data read from the device.

Return type Sequence[Union[int, float]]

query_delay = 0.0

read(*termination: Optional[str] = None*, *encoding: Optional[str] = None*) → str

Read a string from the device.

Reading stops when the device stops sending (e.g. by setting appropriate bus lines), or the termination characters sequence was detected. Attention: Only the last character of the termination characters is really used to stop reading, however, the whole sequence is compared to the ending of the read string message. If they don't match, a warning is issued.

Parameters

- **termination** (*Optional[str], optional*) – Alternative character termination to use. If None, the value of write_termination is used. Defaults to None.
- **encoding** (*Optional[str], optional*) – Alternative encoding to use to turn bytes into str. If None, the value of encoding is used. Defaults to None.

Returns Message read from the instrument and decoded.

Return type str

read_ascii_values(*converter: Union[typing_extensions.Literal['s', 'b', 'c', 'd', 'o', 'x', 'X', 'e', 'E', 'f', 'F', 'g', 'G'][s, b, c, d, o, x, X, e, E, f, F, g, G], Callable[[str], Any]] = 'f', separator: Union[str, Callable[[str], Iterable[str]]] = ',', container: Union[Type[CT_co], Callable[[Iterable[T_co]], Sequence[T_co]]] = <class 'list'>*) → Sequence[T_co]

Read values from the device in ascii format returning an iterable of values.

Parameters **converter** (*ASCII_CONVERTER, optional*) – Str format of function to convert each value. Default to "f".

separator [Union[str, Callable[[str], Iterable[str]]]] str or callable used to split the data into individual elements. If a str is given, data.split(separator) is used. Default to ",".

container [Union[Type, Callable[[Iterable], Sequence]], optional] Container type to use for the output data. Possible values are: list, tuple, np.ndarray, etc, Default to list.

Returns Parsed data.

Return type Sequence

read_binary_values(*datatype: typing_extensions.Literal['s', 'b', 'B', 'h', 'H', 'i', 'I', 'l', 'L', 'q', 'Q', 'f', 'd'][s, b, B, h, H, i, I, l, L, q, Q, f, d] = 'f', is_big_endian: bool = False, container: Union[Type[CT_co], Callable[[Iterable[T_co]], Sequence[T_co]]] = <class 'list'>, header_fmt: typing_extensions.Literal['ieee', 'hp', 'empty'][ieee, hp, empty] = 'ieee', expect_termination: bool = True, data_points: int = 0, chunk_size: Optional[int] = None*) → Sequence[Union[int, float]]

Read values from the device in binary format returning an iterable of values.

Parameters

- **datatype** (*BINARY_DATATYPES, optional*) – Format string for a single element. See struct module. 'f' by default.
- **is_big_endian** (*bool, optional*) – Are the data in big or little endian order. Defaults to False.
- **container** (*Union[Type, Callable[[Iterable], Sequence]], optional*) – Container type to use for the output data. Possible values are: list, tuple, np.ndarray, etc, Default to list.
- **header_fmt** (*util.BINARY_HEADERS, optional*) – Format of the header prefixing the data. Defaults to 'ieee'.
- **expect_termination** (*bool, optional*) – When set to False, the expected length of the binary values block does not account for the final termination character (the read termination). Defaults to True.
- **data_points** (*int, optional*) – Number of points expected in the block. This is used only if the instrument does not report it itself. This will be converted in a number of bytes based on the datatype. Defaults to 0.
- **chunk_size** (*int, optional*) – Size of the chunks to read from the device. Using larger chunks may be faster for large amount of data.

Returns Data read from the device.

Return type Sequence[Union[int, float]]

read_bytes(*count: int, chunk_size: Optional[int] = None, break_on_termchar: bool = False*) → bytes

Read a certain number of bytes from the instrument.

Parameters

- **count** (*int*) – The number of bytes to read from the instrument.
- **chunk_size** (*Optional[int], optional*) – The chunk size to use to perform the reading. If count > chunk_size multiple low level operations will be performed. Defaults to None, meaning the resource wide set value is set.
- **break_on_termchar** (*bool, optional*) – Should the reading stop when a termination character is encountered or when the message ends. Defaults to False.

Returns Bytes read from the instrument.

Return type bytes

read_raw(*size: Optional[int] = None*) → bytes

Read the unmodified string sent from the instrument to the computer.

In contrast to read(), no termination characters are stripped.

Parameters **size** (*Optional[int], optional*) – The chunk size to use to perform the reading. Defaults to None, meaning the resource wide set value is set.

Returns Bytes read from the instrument.

Return type bytes

read_stb() → int
Service request status register.

read_termination
Read termination character.

read_termination_context(*new_termination: str*) → Iterator[T_co]

classmethod register(*interface_type: pyvisa.constants.InterfaceType*, *resource_class: str*) → Callable[[Type[T]], Type[T]]
Create a decorator to register a class.

The class is associated to an interface type, resource class pair.

Parameters

- **interface_type** (constants.InterfaceType) – Interface type for which to register a wrapper class.
- **resource_class** (*str*) – Resource class for which to register a wrapper class.

Returns Decorator registering the class. Raises TypeError if some VISA attributes are missing on the registered class.

Return type Callable[[Type[T]], Type[T]]

remote_enabled
Current state of the GPIB REN (Remote ENable) interface line. :VISA Attribute: VI_ATTR_GPIB_REN_STATE (1073676673) :type: :class:pyvisa.constants.LineState

resource_class
Resource class as defined by the canonical resource name.

Possible values are: INSTR, INTFC, SOCKET, RAW...

VISA Attribute VI_ATTR_RSRC_CLASS (3221159937)

resource_info
Get the extended information of this resource.

resource_manufacturer_name
Manufacturer name of the vendor that implemented the VISA library.

This attribute is not related to the device manufacturer attributes.

Note The value of this attribute is for display purposes only and not for programmatic decisions, as the value can differ between VISA implementations and/or revisions.

VISA Attribute VI_ATTR_RSRC_MANF_NAME (3221160308)

resource_name
Unique identifier for a resource compliant with the address structure. :VISA Attribute: VI_ATTR_RSRC_NAME (3221159938)

secondary_address
Secondary address of the GPIB device used by the given session.

For the GPIB INTFC Resource, this attribute is Read-Write.

VISA Attribute VI_ATTR_GPIB_SECONDARY_ADDR (1073676659)

Type int

Range 0 <= value <= 30 or in [65535]

send_command(*data: bytes*) → Tuple[int, pyvisa.constants.StatusCode]
Write GPIB command bytes on the bus.

Corresponds to viGpibCommand function of the VISA library.

Parameters **data** (*bytes*) – Command to write.

Returns

- *int* – Number of bytes written
- *constants.StatusCode* – Return value of the library call.

send_end
Should END be asserted during the transfer of the last byte of the buffer. :VISA Attribute: VI_ATTR_SEND_END_EN (1073676310) :type: bool

send_ifc() → pyvisa.constants.StatusCode
Pulse the interface clear line (IFC) for at least 100 microseconds.

Corresponds to viGpibSendIFC function of the VISA library.

session
Resource session handle.

Raises errors.InvalidSession – Raised if session is closed.

set_visa_attribute(*name: pyvisa.constants.ResourceAttribute*, *state: Any*) → pyvisa.constants.StatusCode
Set the state of an attribute.

One should prefer the dedicated descriptor for often used attributes since those perform checks and automatic conversion on the value.

Parameters

- **name** (*constants.ResourceAttribute*) – Attribute for which the state is to be modified.
- **state** (*Any*) – The state of the attribute to be set for the specified object.

Returns Return value of the library call.

Return type *constants.StatusCode*

spec_version
Version of the VISA specification to which the implementation is compliant.

The format of the value has the upper 12 bits as the major number of the version, the next lower 12 bits as the minor number of the version, and the lowest 8 bits as the sub-minor number of the version. The current VISA specification defines the value to be 00300000h.

VISA Attribute VI_ATTR_RSRC_SPEC_VERSION (1073676656)

Type int

Range 0 <= value <= 4294967295

stb
Service request status register.

timeout
Timeout in milliseconds for all resource I/O operations.

This value is used when accessing the device associated with the given session.

Special values:

- **immediate (VI_TMO_IMMEDIATE): 0** (for convenience, any value smaller than 1 is considered as 0)
- **infinite (VI_TMO_INFINITE): float('+inf')** (for convenience, None is considered as `float('+inf')`)

To set an **infinite** timeout, you can also use:

```
>>> del instrument.timeout
```

A timeout value of VI_TMO_IMMEDIATE means that operations should never wait for the device to respond. A timeout value of VI_TMO_INFINITE disables the timeout mechanism.

VISA Attribute VI_ATTR_TMO_VALUE (1073676314)

Type int

Range 0 <= value <= 4294967295

uninstall_handler(*event_type: pyvisa.constants.EventType, handler: Callable[[NewType.<locals>.new_type, pyvisa.constants.EventType, NewType.<locals>.new_type, Any], None], user_handle=None*) → None
Uninstalls handlers for events in this resource.

Parameters

- **event_type** (*constants.EventType*) – Logical event identifier.
- **handler** (*VISAHandler*) – Handler function to be uninstalled by a client application.
- **user_handle** (*Any*) – The user handle returned by install_handler.

unlock() → None
Relinquishes a lock for the specified resource.

visa_attributes_classes = {<class 'pyvisa.attributes.AttrVI_ATTR_INTF_INST_NAME'>, <cl

wait_on_event(*in_event_type: pyvisa.constants.EventType, timeout: int, capture_timeout: bool = False*) → pyvisa.resources.resource.WaitResponse
Waits for an occurrence of the specified event in this resource.

in_event_type [constants.EventType] Logical identifier of the event(s) to wait for.

timeout [int] Absolute time period in time units that the resource shall wait for a specified event to occur before returning the time elapsed error. The time unit is in milliseconds. None means waiting forever if necessary.

capture_timeout [bool, optional] When True will not produce a VisaIOError(VI_ERROR_TMO) but instead return a WaitResponse with timed_out=True.

Returns Object that contains event_type, context and ret value.

Return type WaitResponse

wrap_handler(*callable: Callable[[Resource, pyvisa.events.Event, Any], None]*) → Callable[[NewType.<locals>.new_type, pyvisa.constants.EventType, NewType.<locals>.new_type, Any], None]

Wrap an event handler to provide the signature expected by VISA.

The handler is expected to have the following signature: handler(resource: Resource, event: Event, user_handle: Any) -> None.

The wrapped handler should be used only to handle events on the resource used to wrap the handler.

write(*message: str*, *termination: Optional[str] = None*, *encoding: Optional[str] = None*) → int

Write a string message to the device.

The write_termination is always appended to it.

Parameters

- **message** (*str*) – The message to be sent.
- **termination** (*Optional[str], optional*) – Alternative character termination to use. If None, the value of write_termination is used. Defaults to None.
- **encoding** (*Optional[str], optional*) – Alternative encoding to use to turn str into bytes. If None, the value of encoding is used. Defaults to None.

Returns Number of bytes written.

Return type int

write_ascii_values(*message: str, values: Sequence[Any], converter: Union[typing_extensions.Literal['s', 'b', 'c', 'd', 'o', 'x', 'X', 'e', 'E', 'f', 'F', 'g', 'G'][s, b, c, d, o, x, X, e, E, f, F, g, G], Callable[[str], Any]] = 'f', separator: Union[str, Callable[[Iterable[str]], str]] = ',', termination: Optional[str] = None, encoding: Optional[str] = None*)

Write a string message to the device followed by values in ascii format.

The write_termination is always appended to it.

Parameters

- **message** (*str*) – Header of the message to be sent.
- **values** (*Sequence[Any]*) – Data to be writen to the device.
- **converter** (*Union[str, Callable[[Any], str]], optional*) – Str formatting codes or function used to convert each value. Defaults to "f".
- **separator** (*Union[str, Callable[[Iterable[str]], str]], optional*) – Str or callable that join the values in a single str. If a str is given, separator.join(values) is used. Defaults to ','
- **termination** (*Optional[str], optional*) – Alternative character termination to use. If None, the value of write_termination is used. Defaults to None.
- **encoding** (*Optional[str], optional*) – Alternative encoding to use to turn str into bytes. If None, the value of encoding is used. Defaults to None.

Returns Number of bytes written.

Return type int

write_binary_values(*message: str, values: Sequence[Any], datatype: typing_extensions.Literal['s', 'b', 'B', 'h', 'H', 'i', 'I', 'l', 'L', 'q', 'Q', 'f', 'd'][s, b, B, h, H, i, I, l, L, q, Q, f, d] = 'f', is_big_endian: bool = False, termination: Optional[str] = None, encoding: Optional[str] = None, header_fmt: typing_extensions.Literal['ieee', 'hp', 'empty'][ieee, hp, empty] = 'ieee'*)

Write a string message to the device followed by values in binary format.

The write_termination is always appended to it.

Parameters

- **message** (*str*) – The header of the message to be sent.
- **values** (*Sequence[Any]*) – Data to be written to the device.
- **datatype** (*util.BINARY_DATATYPES, optional*) – The format string for a single element. See struct module.
- **is_big_endian** (*bool, optional*) – Are the data in big or little endian order.
- **termination** (*Optional[str], optional*) – Alternative character termination to use. If None, the value of write_termination is used. Defaults to None.
- **encoding** (*Optional[str], optional*) – Alternative encoding to use to turn str into bytes. If None, the value of encoding is used. Defaults to None.
- **header_fmt** (*util.BINARY_HEADERS*) – Format of the header prefixing the data.

Returns Number of bytes written.

Return type int

write_raw(*message: bytes*) → int

Write a byte message to the device.

Parameters **message** (*bytes*) – The message to be sent.

Returns Number of bytes written

Return type int

write_termination

Write termination character.

class pyvisa.resources.**FirewireInstrument**(*resource_manager: pyvisa.highlevel.ResourceManager, resource_name: str*)

Communicates with devices of type VXI::VXI logical address[::INSTR]

More complex resource names can be specified with the following grammar: VXI[board]::VXI logical address[::INSTR]

Do not instantiate directly, use *pyvisa.highlevel.ResourceManager.open_resource()*.

before_close() → None

Called just before closing an instrument.

clear() → None

Clear this resource.

close() → None

Closes the VISA session and marks the handle as invalid.

disable_event(*event_type: pyvisa.constants.EventType*, *mechanism: pyvisa.constants.EventMechanism*) → None
Disable notification for an event type(s) via the specified mechanism(s).

Parameters

- **event_type** (*constants.EventType*) – Logical event identifier.
- **mechanism** (*constants.EventMechanism*) – Specifies event handling mechanisms to be disabled.

discard_events(*event_type: pyvisa.constants.EventType*, *mechanism: pyvisa.constants.EventMechanism*) → None
Discards event occurrences for an event type and mechanism in this resource.

Parameters

- **event_type** (*constants.EventType*) – Logical event identifier.
- **mechanism** (*constants.EventMechanism*) – Specifies event handling mechanisms to be disabled.

enable_event(*event_type: pyvisa.constants.EventType*, *mechanism: pyvisa.constants.EventMechanism*, *context: None = None*) → None
Enable event occurrences for specified event types and mechanisms in this resource.

Parameters

- **event_type** (*constants.EventType*) – Logical event identifier.
- **mechanism** (*constants.EventMechanism*) – Specifies event handling mechanisms to be enabled
- **context** (*None*) – Not currently used, leave as None.

get_visa_attribute(*name: pyvisa.constants.ResourceAttribute*) → Any
Retrieves the state of an attribute in this resource.

One should prefer the dedicated descriptor for often used attributes since those perform checks and automatic conversion on the value.

Parameters **name** (*constants.ResourceAttribute*) – Resource attribute for which the state query is made.

Returns The state of the queried attribute for a specified resource.

Return type Any

ignore_warning(**warnings_constants*) → AbstractContextManager[T_co]
Ignoring warnings context manager for the current resource.

Parameters **warnings_constants** (constants.StatusCode) – Constants identifying the warnings to ignore.

implementation_version
Resource version that identifies the revisions or implementations of a resource.

This attribute value is defined by the individual manufacturer and increments with each new revision. The format of the value has the upper 12 bits as the major number of the version, the next lower 12 bits as the minor number of the version, and the lowest 8 bits as the sub-minor number of the version.

VISA Attribute VI_ATTR_RSRC_IMPL_VERSION (1073676291)

Type int

Range 0 <= value <= 4294967295

install_handler(*event_type: pyvisa.constants.EventType, handler: Callable[[NewType.<locals>.new_type, pyvisa.constants.EventType, NewType.<locals>.new_type, Any], None], user_handle=None*) → Any

Install handlers for event callbacks in this resource.

Parameters

- **event_type** (*constants.EventType*) – Logical event identifier.
- **handler** (*VISAHandler*) – Handler function to be installed by a client application.
- **user_handle** – A value specified by an application that can be used for identifying handlers uniquely for an event type. Depending on the backend they may be restriction on the possible values. Look at the backend *install_visa_handler* for more details.

Returns User handle in a format amenable to the backend. This is this representation of the handle that should be used when unistalling a handler.

Return type Any

interface_number

Board number for the given interface. :VISA Attribute: VI_ATTR_INTF_NUM (1073676662) :type: int :range: 0 <= value <= 65535

interface_type

Interface type of the given session. :VISA Attribute: VI_ATTR_INTF_TYPE (1073676657) :type: :class:pyvisa.constants.InterfaceType

last_status

Last status code for this session.

lock(*timeout: Union[float, typing_extensions.Literal['default'][default]] = 'default', requested_key: Optional[str] = None*) → str

Establish a shared lock to the resource.

Parameters

- **timeout** (*Union[float, Literal["default"]], optional*) – Absolute time period (in milliseconds) that a resource waits to get unlocked by the locking session before returning an error. Defaults to "default" which means use self.timeout.
- **requested_key** (*Optional[str], optional*) – Access key used by another session with which you want your session to share a lock or None to generate a new shared access key.

Returns A new shared access key if requested_key is None, otherwise, same value as the requested_key

Return type str

lock_context(*timeout: Union[float, typing_extensions.Literal['default'][default]] = 'default', requested_key: Optional[str] = 'exclusive'*) → Iterator[Optional[str]]

A context that locks

Parameters

- **timeout** (*Union[float, Literal["default"]], optional*) – Absolute time period (in milliseconds) that a resource waits to get unlocked by the locking session before returning an error. Defaults to "default" which means use self.timeout.

- **requested_key** (*Optional[str], optional*) – When using default of 'exclusive' the lock is an exclusive lock. Otherwise it is the access key for the shared lock or None to generate a new shared access key.

Yields *Optional[str]* – The access_key if applicable.

lock_excl(*timeout: Union[float, typing_extensions.Literal['default']][default]] = 'default'*) → None
Establish an exclusive lock to the resource.

Parameters **timeout** (*Union[float, Literal["default"]], optional*) – Absolute time period (in milliseconds) that a resource waits to get unlocked by the locking session before returning an error. Defaults to "default" which means use self.timeout.

lock_state
Current locking state of the resource.

The resource can be unlocked, locked with an exclusive lock, or locked with a shared lock.

VISA Attribute VI_ATTR_RSRC_LOCK_STATE (1073676292)

Type :class:pyvisa.constants.AccessModes

move_in(*space: pyvisa.constants.AddressSpace, offset: int, length: int, width: pyvisa.constants.DataWidth, extended: bool = False*) → List[int]
Move a block of data to local memory from the given address space and offset.

Corresponds to viMoveIn* functions of the VISA library.

Parameters

- **space** (*constants.AddressSpace*) – Address space from which to move the data.
- **offset** (*int*) – Offset (in bytes) of the address or register from which to read.
- **length** (*int*) – Number of elements to transfer, where the data width of the elements to transfer is identical to the source data width.
- **width** (*Union[Literal[8, 16, 32, 64], constants.DataWidth]*) – Number of bits to read per element.
- **extended** (*bool, optional*) – Use 64 bits offset independent of the platform, by default False.

Returns

- **data** (*List[int]*) – Data read from the bus
- **status_code** (*constants.StatusCode*) – Return value of the library call.

Raises ValueError – Raised if an invalid width is specified.

move_out(*space: pyvisa.constants.AddressSpace, offset: int, length: int, data: Iterable[int], width: pyvisa.constants.DataWidth, extended: bool = False*) → pyvisa.constants.StatusCode
Move a block of data from local memory to the given address space and offset.

Corresponds to viMoveOut* functions of the VISA library.

Parameters

- **space** (*constants.AddressSpace*) – Address space into which move the data.
- **offset** (*int*) – Offset (in bytes) of the address or register from which to read.
- **length** (*int*) – Number of elements to transfer, where the data width of the elements to transfer is identical to the source data width.

- **data** (*Iterable[int]*) – Data to write to bus.
- **width** (*Union[Literal[8, 16, 32, 64], constants.DataWidth]*) – Number of bits to per element.
- **extended** (*bool, optional*) – Use 64 bits offset independent of the platform, by default False.

Returns Return value of the library call.

Return type *constants.StatusCode*

Raises ValueError – Raised if an invalid width is specified.

open(*access_mode: pyvisa.constants.AccessModes = <AccessModes.no_lock: 0>*, *open_timeout: int = 5000*) → None

Opens a session to the specified resource.

Parameters

- **access_mode** (*constants.AccessModes, optional*) – Specifies the mode by which the resource is to be accessed. Defaults to constants.AccessModes.no_lock.
- **open_timeout** (*int, optional*) – If the access_mode parameter requests a lock, then this parameter specifies the absolute time period (in milliseconds) that the resource waits to get unlocked before this operation returns an error. Defaults to 5000.

read_memory(*space: pyvisa.constants.AddressSpace*, *offset: int*, *width: pyvisa.constants.DataWidth*, *extended: bool = False*) → int

Read a value from the specified memory space and offset.

Parameters

- **space** (*constants.AddressSpace*) – Specifies the address space from which to read.
- **offset** (*int*) – Offset (in bytes) of the address or register from which to read.
- **width** (*Union[Literal[8, 16, 32, 64], constants.DataWidth]*) – Number of bits to read (8, 16, 32 or 64).
- **extended** (*bool, optional*) – Use 64 bits offset independent of the platform.

Returns **data** – Data read from memory

Return type int

Raises ValueError – Raised if an invalid width is specified.

classmethod register(*interface_type: pyvisa.constants.InterfaceType*, *resource_class: str*) → Callable[[Type[T]], Type[T]]

Create a decorator to register a class.

The class is associated to an interface type, resource class pair.

Parameters

- **interface_type** (*constants.InterfaceType*) – Interface type for which to register a wrapper class.
- **resource_class** (*str*) – Resource class for which to register a wrapper class.

Returns Decorator registering the class. Raises TypeError if some VISA attributes are missing on the registered class.

Return type Callable[[Type[T]], Type[T]]

resource_class
Resource class as defined by the canonical resource name.

Possible values are: INSTR, INTFC, SOCKET, RAW...

VISA Attribute VI_ATTR_RSRC_CLASS (3221159937)

resource_info
Get the extended information of this resource.

resource_manufacturer_name
Manufacturer name of the vendor that implemented the VISA library.

This attribute is not related to the device manufacturer attributes.

Note The value of this attribute is for display purposes only and not for programmatic decisions, as the value can differ between VISA implementations and/or revisions.

VISA Attribute VI_ATTR_RSRC_MANF_NAME (3221160308)

resource_name
Unique identifier for a resource compliant with the address structure. :VISA Attribute: VI_ATTR_RSRC_NAME (3221159938)

session
Resource session handle.

Raises `errors.InvalidSession` – Raised if session is closed.

set_visa_attribute(*name: pyvisa.constants.ResourceAttribute*, *state: Any*) → pyvisa.constants.StatusCode
Set the state of an attribute.

One should prefer the dedicated descriptor for often used attributes since those perform checks and automatic conversion on the value.

Parameters

- **name** (*constants.ResourceAttribute*) – Attribute for which the state is to be modified.
- **state** (*Any*) – The state of the attribute to be set for the specified object.

Returns Return value of the library call.

Return type *constants.StatusCode*

spec_version
Version of the VISA specification to which the implementation is compliant.

The format of the value has the upper 12 bits as the major number of the version, the next lower 12 bits as the minor number of the version, and the lowest 8 bits as the sub-minor number of the version. The current VISA specification defines the value to be 00300000h.

VISA Attribute VI_ATTR_RSRC_SPEC_VERSION (1073676656)

Type int

Range 0 <= value <= 4294967295

timeout
Timeout in milliseconds for all resource I/O operations.

This value is used when accessing the device associated with the given session.

Special values:

- **immediate (`VI_TMO_IMMEDIATE`): 0** (for convenience, any value smaller than 1 is considered as 0)
- **infinite (`VI_TMO_INFINITE`): `float('+inf')`** (for convenience, None is considered as `float('+inf')`)

To set an **infinite** timeout, you can also use:

```
>>> del instrument.timeout
```

A timeout value of VI_TMO_IMMEDIATE means that operations should never wait for the device to respond. A timeout value of VI_TMO_INFINITE disables the timeout mechanism.

VISA Attribute VI_ATTR_TMO_VALUE (1073676314)

Type int

Range 0 <= value <= 4294967295

uninstall_handler(*event_type: pyvisa.constants.EventType, handler: Callable[[NewType.<locals>.new_type, pyvisa.constants.EventType, NewType.<locals>.new_type, Any], None], user_handle=None*) → None

Uninstalls handlers for events in this resource.

Parameters

- **event_type** (*constants.EventType*) – Logical event identifier.
- **handler** (*VISAHandler*) – Handler function to be uninstalled by a client application.
- **user_handle** (*Any*) – The user handle returned by install_handler.

unlock() → None

Relinquishes a lock for the specified resource.

visa_attributes_classes = {<class 'pyvisa.attributes.AttrVI_ATTR_RSRC_NAME'>, <class '

wait_on_event(*in_event_type: pyvisa.constants.EventType, timeout: int, capture_timeout: bool = False*) → pyvisa.resources.resource.WaitResponse

Waits for an occurrence of the specified event in this resource.

in_event_type [constants.EventType] Logical identifier of the event(s) to wait for.

timeout [int] Absolute time period in time units that the resource shall wait for a specified event to occur before returning the time elapsed error. The time unit is in milliseconds. None means waiting forever if necessary.

capture_timeout [bool, optional] When True will not produce a VisaIOError(VI_ERROR_TMO) but instead return a WaitResponse with timed_out=True.

Returns Object that contains event_type, context and ret value.

Return type WaitResponse

wrap_handler(*callable: Callable[[Resource, pyvisa.events.Event, Any], None]*) → Callable[[NewType.<locals>.new_type, pyvisa.constants.EventType, NewType.<locals>.new_type, Any], None]

Wrap an event handler to provide the signature expected by VISA.

The handler is expected to have the following signature: handler(resource: Resource, event: Event, user_handle: Any) -> None.

The wrapped handler should be used only to handle events on the resource used to wrap the handler.

write_memory(*space: pyvisa.constants.AddressSpace*, *offset: int*, *data: int*, *width: pyvisa.constants.DataWidth*, *extended: bool = False*) → pyvisa.constants.StatusCode

Write a value to the specified memory space and offset.

Parameters

- **space** (*constants.AddressSpace*) – Specifies the address space.
- **offset** (*int*) – Offset (in bytes) of the address or register from which to read.
- **data** (*int*) – Data to write to bus.
- **width** (*Union[Literal[8, 16, 32, 64], constants.DataWidth]*) – Number of bits to read.
- **extended** (*bool, optional*) – Use 64 bits offset independent of the platform, by default False.

Returns Return value of the library call.

Return type *constants.StatusCode*

Raises ValueError – Raised if an invalid width is specified.

class pyvisa.resources.**PXIInstrument**(*resource_manager: pyvisa.highlevel.ResourceManager*, *resource_name: str*)

Communicates with to devices of type PXI::<device>[::INSTR]

More complex resource names can be specified with the following grammar:

PXI[bus]::device[::function][::INSTR]

or: PXI[interface]::bus-device[.function][::INSTR]

or: PXI[interface]::CHASSISchassis number::SLOTslot number[::FUNCfunction][::INSTR]

Do not instantiate directly, use *pyvisa.highlevel.ResourceManager.open_resource()*.

allow_dma

Should I/O accesses use DMA (True) or Programmed I/O (False).

In some implementations, this attribute may have global effects even though it is documented to be a local attribute. Since this affects performance and not functionality, that behavior is acceptable.

VISA Attribute VI_ATTR_DMA_ALLOW_EN (1073676318)

Type bool

before_close() → None

Called just before closing an instrument.

clear() → None

Clear this resource.

close() → None

Closes the VISA session and marks the handle as invalid.

destination_increment

Number of elements by which to increment the destination offset after a transfer.

The default value of this attribute is 1 (that is, the destination address will be incremented by 1 after each transfer), and the viMoveOutXX() operations move into consecutive elements. If this attribute is set to 0, the viMoveOutXX() operations will always write to the same element, essentially treating the destination as a FIFO register.

VISA Attribute VI_ATTR_DEST_INCREMENT (1073676353)

Type int

Range 0 <= value <= 1

disable_event(*event_type: pyvisa.constants.EventType*, *mechanism: pyvisa.constants.EventMechanism*) → None

Disable notification for an event type(s) via the specified mechanism(s).

Parameters

- **event_type** (*constants.EventType*) – Logical event identifier.
- **mechanism** (*constants.EventMechanism*) – Specifies event handling mechanisms to be disabled.

discard_events(*event_type: pyvisa.constants.EventType*, *mechanism: pyvisa.constants.EventMechanism*) → None

Discards event occurrences for an event type and mechanism in this resource.

Parameters

- **event_type** (*constants.EventType*) – Logical event identifier.
- **mechanism** (*constants.EventMechanism*) – Specifies event handling mechanisms to be disabled.

enable_event(*event_type: pyvisa.constants.EventType*, *mechanism: pyvisa.constants.EventMechanism*, *context: None = None*) → None

Enable event occurrences for specified event types and mechanisms in this resource.

Parameters

- **event_type** (*constants.EventType*) – Logical event identifier.
- **mechanism** (*constants.EventMechanism*) – Specifies event handling mechanisms to be enabled
- **context** (*None*) – Not currently used, leave as None.

get_visa_attribute(*name: pyvisa.constants.ResourceAttribute*) → Any

Retrieves the state of an attribute in this resource.

One should prefer the dedicated descriptor for often used attributes since those perform checks and automatic conversion on the value.

Parameters **name** (*constants.ResourceAttribute*) – Resource attribute for which the state query is made.

Returns The state of the queried attribute for a specified resource.

Return type Any

ignore_warning(**warnings_constants*) → AbstractContextManager[T_co]

Ignoring warnings context manager for the current resource.

Parameters **warnings_constants** (constants.StatusCode) – Constants identifying the warnings to ignore.

implementation_version
Resource version that identifies the revisions or implementations of a resource.

This attribute value is defined by the individual manufacturer and increments with each new revision. The format of the value has the upper 12 bits as the major number of the version, the next lower 12 bits as the minor number of the version, and the lowest 8 bits as the sub-minor number of the version.

VISA Attribute VI_ATTR_RSRC_IMPL_VERSION (1073676291)

Type int

Range 0 <= value <= 4294967295

install_handler(*event_type: pyvisa.constants.EventType, handler: Callable[[NewType.<locals>.new_type, pyvisa.constants.EventType, NewType.<locals>.new_type, Any], None], user_handle=None*) → Any
Install handlers for event callbacks in this resource.

Parameters

- **event_type** (*constants.EventType*) – Logical event identifier.
- **handler** (*VISAHandler*) – Handler function to be installed by a client application.
- **user_handle** – A value specified by an application that can be used for identifying handlers uniquely for an event type. Depending on the backend they may be restriction on the possible values. Look at the backend *install_visa_handler* for more details.

Returns User handle in a format amenable to the backend. This is this representation of the handle that should be used when unistalling a handler.

Return type Any

interface_number
Board number for the given interface. :VISA Attribute: VI_ATTR_INTF_NUM (1073676662) :type: int :range: 0 <= value <= 65535

interface_type
Interface type of the given session. :VISA Attribute: VI_ATTR_INTF_TYPE (1073676657) :type: :class:pyvisa.constants.InterfaceType

last_status
Last status code for this session.

lock(*timeout: Union[float, typing_extensions.Literal['default'][default]] = 'default', requested_key: Optional[str] = None*) → str
Establish a shared lock to the resource.

Parameters

- **timeout** (*Union[float, Literal["default"]], optional*) – Absolute time period (in milliseconds) that a resource waits to get unlocked by the locking session before returning an error. Defaults to "default" which means use self.timeout.
- **requested_key** (*Optional[str], optional*) – Access key used by another session with which you want your session to share a lock or None to generate a new shared access key.

Returns A new shared access key if requested_key is None, otherwise, same value as the requested_key

Return type str

lock_context(*timeout: Union[float, typing_extensions.Literal['default'][default]] = 'default', requested_key: Optional[str] = 'exclusive'*) → Iterator[Optional[str]]
A context that locks

Parameters

- **timeout** (*Union[float, Literal["default"]], optional*) – Absolute time period (in milliseconds) that a resource waits to get unlocked by the locking session before returning an error. Defaults to "default" which means use self.timeout.
- **requested_key** (*Optional[str], optional*) – When using default of 'exclusive' the lock is an exclusive lock. Otherwise it is the access key for the shared lock or None to generate a new shared access key.

Yields *Optional[str]* – The access_key if applicable.

lock_excl(*timeout: Union[float, typing_extensions.Literal['default'][default]] = 'default'*) → None
Establish an exclusive lock to the resource.

Parameters **timeout** (*Union[float, Literal["default"]], optional*) – Absolute time period (in milliseconds) that a resource waits to get unlocked by the locking session before returning an error. Defaults to "default" which means use self.timeout.

lock_state
Current locking state of the resource.

The resource can be unlocked, locked with an exclusive lock, or locked with a shared lock.

VISA Attribute VI_ATTR_RSRC_LOCK_STATE (1073676292)

Type :class:pyvisa.constants.AccessModes

manufacturer_id
Manufacturer identification number of the device.

manufacturer_name
Manufacturer name.

model_code
Model code for the device.

model_name
Model name of the device.

move_in(*space: pyvisa.constants.AddressSpace, offset: int, length: int, width: pyvisa.constants.DataWidth, extended: bool = False*) → List[int]
Move a block of data to local memory from the given address space and offset.

Corresponds to viMoveIn* functions of the VISA library.

Parameters

- **space** (*constants.AddressSpace*) – Address space from which to move the data.
- **offset** (*int*) – Offset (in bytes) of the address or register from which to read.
- **length** (*int*) – Number of elements to transfer, where the data width of the elements to transfer is identical to the source data width.
- **width** (*Union[Literal[8, 16, 32, 64], constants.DataWidth]*) – Number of bits to read per element.
- **extended** (*bool, optional*) – Use 64 bits offset independent of the platform, by default False.

Returns

- **data** (*List[int]*) – Data read from the bus
- **status_code** (*constants.StatusCode*) – Return value of the library call.

Raises `ValueError` – Raised if an invalid width is specified.

move_out(*space: pyvisa.constants.AddressSpace, offset: int, length: int, data: Iterable[int], width: pyvisa.constants.DataWidth, extended: bool = False*) → pyvisa.constants.StatusCode

Move a block of data from local memory to the given address space and offset.

Corresponds to viMoveOut* functions of the VISA library.

Parameters

- **space** (*constants.AddressSpace*) – Address space into which move the data.
- **offset** (*int*) – Offset (in bytes) of the address or register from which to read.
- **length** (*int*) – Number of elements to transfer, where the data width of the elements to transfer is identical to the source data width.
- **data** (*Iterable[int]*) – Data to write to bus.
- **width** (*Union[Literal[8, 16, 32, 64], constants.DataWidth]*) – Number of bits to per element.
- **extended** (*bool, optional*) – Use 64 bits offset independent of the platform, by default False.

Returns Return value of the library call.

Return type *constants.StatusCode*

Raises `ValueError` – Raised if an invalid width is specified.

open(*access_mode: pyvisa.constants.AccessModes = <AccessModes.no_lock: 0>, open_timeout: int = 5000*) → None

Opens a session to the specified resource.

Parameters

- **access_mode** (`constants.AccessModes, optional`) – Specifies the mode by which the resource is to be accessed. Defaults to constants.AccessModes.no_lock.
- **open_timeout** (*int, optional*) – If the `access_mode` parameter requests a lock, then this parameter specifies the absolute time period (in milliseconds) that the resource waits to get unlocked before this operation returns an error. Defaults to 5000.

read_memory(*space: pyvisa.constants.AddressSpace, offset: int, width: pyvisa.constants.DataWidth, extended: bool = False*) → int

Read a value from the specified memory space and offset.

Parameters

- **space** (*constants.AddressSpace*) – Specifies the address space from which to read.
- **offset** (*int*) – Offset (in bytes) of the address or register from which to read.
- **width** (*Union[Literal[8, 16, 32, 64], constants.DataWidth]*) – Number of bits to read (8, 16, 32 or 64).
- **extended** (*bool, optional*) – Use 64 bits offset independent of the platform.

Returns **data** – Data read from memory

Return type int

Raises `ValueError` – Raised if an invalid width is specified.

classmethod register(*interface_type: pyvisa.constants.InterfaceType*, *resource_class: str*) → Callable[[Type[T]], Type[T]]

Create a decorator to register a class.

The class is associated to an interface type, resource class pair.

Parameters

- **interface_type** (`constants.InterfaceType`) – Interface type for which to register a wrapper class.
- **resource_class** (*str*) – Resource class for which to register a wrapper class.

Returns Decorator registering the class. Raises TypeError if some VISA attributes are missing on the registered class.

Return type Callable[[Type[T]], Type[T]]

resource_class

Resource class as defined by the canonical resource name.

Possible values are: INSTR, INTFC, SOCKET, RAW...

VISA Attribute VI_ATTR_RSRC_CLASS (3221159937)

resource_info

Get the extended information of this resource.

resource_manufacturer_name

Manufacturer name of the vendor that implemented the VISA library.

This attribute is not related to the device manufacturer attributes.

Note The value of this attribute is for display purposes only and not for programmatic decisions, as the value can differ between VISA implementations and/or revisions.

VISA Attribute VI_ATTR_RSRC_MANF_NAME (3221160308)

resource_name

Unique identifier for a resource compliant with the address structure. :VISA Attribute: VI_ATTR_RSRC_NAME (3221159938)

session

Resource session handle.

Raises `errors.InvalidSession` – Raised if session is closed.

set_visa_attribute(*name: pyvisa.constants.ResourceAttribute*, *state: Any*) → pyvisa.constants.StatusCode

Set the state of an attribute.

One should prefer the dedicated descriptor for often used attributes since those perform checks and automatic conversion on the value.

Parameters

- **name** (*constants.ResourceAttribute*) – Attribute for which the state is to be modified.
- **state** (*Any*) – The state of the attribute to be set for the specified object.

Returns Return value of the library call.

Return type *constants.StatusCode*

source_increment

Number of elements by which to increment the source offset after a transfer.

The default value of this attribute is 1 (that is, the source address will be incremented by 1 after each transfer), and the viMoveInXX() operations move from consecutive elements. If this attribute is set to 0, the viMoveInXX() operations will always read from the same element, essentially treating the source as a FIFO register.

VISA Attribute VI_ATTR_SRC_INCREMENT (1073676352)

Type int

Range 0 <= value <= 1

spec_version

Version of the VISA specification to which the implementation is compliant.

The format of the value has the upper 12 bits as the major number of the version, the next lower 12 bits as the minor number of the version, and the lowest 8 bits as the sub-minor number of the version. The current VISA specification defines the value to be 00300000h.

VISA Attribute VI_ATTR_RSRC_SPEC_VERSION (1073676656)

Type int

Range 0 <= value <= 4294967295

timeout

Timeout in milliseconds for all resource I/O operations.

This value is used when accessing the device associated with the given session.

Special values:

- **immediate (`VI_TMO_IMMEDIATE`): 0** (for convenience, any value smaller than 1 is considered as 0)
- **infinite (`VI_TMO_INFINITE`): `float('+inf')`** (for convenience, None is considered as `float('+inf')`)

To set an **infinite** timeout, you can also use:

```
>>> del instrument.timeout
```

A timeout value of VI_TMO_IMMEDIATE means that operations should never wait for the device to respond. A timeout value of VI_TMO_INFINITE disables the timeout mechanism.

VISA Attribute VI_ATTR_TMO_VALUE (1073676314)

Type int

Range 0 <= value <= 4294967295

uninstall_handler(*event_type: pyvisa.constants.EventType, handler: Callable[[NewType.<locals>.new_type, pyvisa.constants.EventType, NewType.<locals>.new_type, Any], None], user_handle=None*) → None

Uninstalls handlers for events in this resource.

Parameters

- **event_type** (*constants.EventType*) – Logical event identifier.
- **handler** (*VISAHandler*) – Handler function to be uninstalled by a client application.
- **user_handle** (*Any*) – The user handle returned by install_handler.

unlock() → None
Relinquishes a lock for the specified resource.

visa_attributes_classes = {<class 'pyvisa.attributes.AttrVI_ATTR_INTF_INST_NAME'>, <cl

wait_on_event(*in_event_type: pyvisa.constants.EventType*, *timeout: int*, *capture_timeout: bool = False*) → pyvisa.resources.resource.WaitResponse
Waits for an occurrence of the specified event in this resource.

in_event_type [constants.EventType] Logical identifier of the event(s) to wait for.

timeout [int] Absolute time period in time units that the resource shall wait for a specified event to occur before returning the time elapsed error. The time unit is in milliseconds. None means waiting forever if necessary.

capture_timeout [bool, optional] When True will not produce a VisaIOError(VI_ERROR_TMO) but instead return a WaitResponse with timed_out=True.

Returns Object that contains event_type, context and ret value.

Return type WaitResponse

wrap_handler(*callable: Callable[[Resource, pyvisa.events.Event, Any], None]*) → Callable[[NewType.<locals>.new_type, pyvisa.constants.EventType, NewType.<locals>.new_type, Any], None]
Wrap an event handler to provide the signature expected by VISA.

The handler is expected to have the following signature: handler(resource: Resource, event: Event, user_handle: Any) -> None.

The wrapped handler should be used only to handle events on the resource used to wrap the handler.

write_memory(*space: pyvisa.constants.AddressSpace*, *offset: int*, *data: int*, *width: pyvisa.constants.DataWidth*, *extended: bool = False*) → pyvisa.constants.StatusCode
Write a value to the specified memory space and offset.

Parameters

- **space** (*constants.AddressSpace*) – Specifies the address space.
- **offset** (*int*) – Offset (in bytes) of the address or register from which to read.
- **data** (*int*) – Data to write to bus.
- **width** (*Union[Literal[8, 16, 32, 64], constants.DataWidth]*) – Number of bits to read.
- **extended** (*bool, optional*) – Use 64 bits offset independent of the platform, by default False.

Returns Return value of the library call.

Return type *constants.StatusCode*

Raises ValueError – Raised if an invalid width is specified.

class pyvisa.resources.**PXIMemory**(*resource_manager: pyvisa.highlevel.ResourceManager*, *resource_name: str*)

Communicates with to devices of type PXI[interface]::MEMACC

Do not instantiate directly, use *pyvisa.highlevel.ResourceManager.open_resource()*.

allow_dma

Should I/O accesses use DMA (True) or Programmed I/O (False).

In some implementations, this attribute may have global effects even though it is documented to be a local attribute. Since this affects performance and not functionality, that behavior is acceptable.

VISA Attribute VI_ATTR_DMA_ALLOW_EN (1073676318)

Type bool

before_close() → None

Called just before closing an instrument.

clear() → None

Clear this resource.

close() → None

Closes the VISA session and marks the handle as invalid.

destination_increment

Number of elements by which to increment the destination offset after a transfer.

The default value of this attribute is 1 (that is, the destination address will be incremented by 1 after each transfer), and the viMoveOutXX() operations move into consecutive elements. If this attribute is set to 0, the viMoveOutXX() operations will always write to the same element, essentially treating the destination as a FIFO register.

VISA Attribute VI_ATTR_DEST_INCREMENT (1073676353)

Type int

Range 0 <= value <= 1

disable_event(*event_type: pyvisa.constants.EventType*, *mechanism: pyvisa.constants.EventMechanism*) → None

Disable notification for an event type(s) via the specified mechanism(s).

Parameters

- **event_type** (*constants.EventType*) – Logical event identifier.
- **mechanism** (*constants.EventMechanism*) – Specifies event handling mechanisms to be disabled.

discard_events(*event_type: pyvisa.constants.EventType*, *mechanism: pyvisa.constants.EventMechanism*) → None

Discards event occurrences for an event type and mechanism in this resource.

Parameters

- **event_type** (*constants.EventType*) – Logical event identifier.
- **mechanism** (*constants.EventMechanism*) – Specifies event handling mechanisms to be disabled.

enable_event(*event_type: pyvisa.constants.EventType*, *mechanism: pyvisa.constants.EventMechanism*, *context: None = None*) → None
Enable event occurrences for specified event types and mechanisms in this resource.

Parameters

- **event_type** (*constants.EventType*) – Logical event identifier.
- **mechanism** (*constants.EventMechanism*) – Specifies event handling mechanisms to be enabled
- **context** (*None*) – Not currently used, leave as None.

get_visa_attribute(*name: pyvisa.constants.ResourceAttribute*) → Any
Retrieves the state of an attribute in this resource.

One should prefer the dedicated descriptor for often used attributes since those perform checks and automatic conversion on the value.

Parameters **name** (*constants.ResourceAttribute*) – Resource attribute for which the state query is made.

Returns The state of the queried attribute for a specified resource.

Return type Any

ignore_warning(**warnings_constants*) → AbstractContextManager[T_co]
Ignoring warnings context manager for the current resource.

Parameters **warnings_constants** (constants.StatusCode) – Constants identifying the warnings to ignore.

implementation_version
Resource version that identifies the revisions or implementations of a resource.

This attribute value is defined by the individual manufacturer and increments with each new revision. The format of the value has the upper 12 bits as the major number of the version, the next lower 12 bits as the minor number of the version, and the lowest 8 bits as the sub-minor number of the version.

VISA Attribute VI_ATTR_RSRC_IMPL_VERSION (1073676291)

Type int

Range 0 <= value <= 4294967295

install_handler(*event_type: pyvisa.constants.EventType, handler: Callable[[NewType.<locals>.new_type, pyvisa.constants.EventType, NewType.<locals>.new_type, Any], None], user_handle=None*) → Any
Install handlers for event callbacks in this resource.

Parameters

- **event_type** (*constants.EventType*) – Logical event identifier.
- **handler** (*VISAHandler*) – Handler function to be installed by a client application.
- **user_handle** – A value specified by an application that can be used for identifying handlers uniquely for an event type. Depending on the backend they may be restriction on the possible values. Look at the backend *install_visa_handler* for more details.

Returns User handle in a format amenable to the backend. This is this representation of the handle that should be used when unistalling a handler.

Return type Any

interface_number
Board number for the given interface. :VISA Attribute: VI_ATTR_INTF_NUM (1073676662) :type: int :range: 0 <= value <= 65535

interface_type
Interface type of the given session. :VISA Attribute: VI_ATTR_INTF_TYPE (1073676657) :type: :class:pyvisa.constants.InterfaceType

last_status
Last status code for this session.

lock(*timeout: Union[float, typing_extensions.Literal['default']][default]] = 'default', requested_key: Optional[str] = None*) → str
Establish a shared lock to the resource.

Parameters

- **timeout** (*Union[float, Literal["default"]], optional*) – Absolute time period (in milliseconds) that a resource waits to get unlocked by the locking session before returning an error. Defaults to "default" which means use self.timeout.
- **requested_key** (*Optional[str], optional*) – Access key used by another session with which you want your session to share a lock or None to generate a new shared access key.

Returns A new shared access key if requested_key is None, otherwise, same value as the requested_key

Return type str

lock_context(*timeout: Union[float, typing_extensions.Literal['default']][default]] = 'default', requested_key: Optional[str] = 'exclusive'*) → Iterator[Optional[str]]
A context that locks

Parameters

- **timeout** (*Union[float, Literal["default"]], optional*) – Absolute time period (in milliseconds) that a resource waits to get unlocked by the locking session before returning an error. Defaults to "default" which means use self.timeout.
- **requested_key** (*Optional[str], optional*) – When using default of 'exclusive' the lock is an exclusive lock. Otherwise it is the access key for the shared lock or None to generate a new shared access key.

Yields *Optional[str]* – The access_key if applicable.

lock_excl(*timeout: Union[float, typing_extensions.Literal['default']][default]] = 'default'*) → None
Establish an exclusive lock to the resource.

Parameters **timeout** (*Union[float, Literal["default"]], optional*) – Absolute time period (in milliseconds) that a resource waits to get unlocked by the locking session before returning an error. Defaults to "default" which means use self.timeout.

lock_state
Current locking state of the resource.

The resource can be unlocked, locked with an exclusive lock, or locked with a shared lock.

VISA Attribute VI_ATTR_RSRC_LOCK_STATE (1073676292)

Type :class:pyvisa.constants.AccessModes

move_in(*space: pyvisa.constants.AddressSpace, offset: int, length: int, width: pyvisa.constants.DataWidth, extended: bool = False*) → List[int]
Move a block of data to local memory from the given address space and offset.

Corresponds to viMoveIn* functions of the VISA library.

Parameters

- **space** (*constants.AddressSpace*) – Address space from which to move the data.
- **offset** (*int*) – Offset (in bytes) of the address or register from which to read.
- **length** (*int*) – Number of elements to transfer, where the data width of the elements to transfer is identical to the source data width.
- **width** (*Union[Literal[8, 16, 32, 64], constants.DataWidth]*) – Number of bits to read per element.
- **extended** (*bool, optional*) – Use 64 bits offset independent of the platform, by default False.

Returns

- **data** (*List[int]*) – Data read from the bus
- **status_code** (*constants.StatusCode*) – Return value of the library call.

Raises ValueError – Raised if an invalid width is specified.

move_out(*space: pyvisa.constants.AddressSpace, offset: int, length: int, data: Iterable[int], width: pyvisa.constants.DataWidth, extended: bool = False*) → pyvisa.constants.StatusCode
Move a block of data from local memory to the given address space and offset.

Corresponds to viMoveOut* functions of the VISA library.

Parameters

- **space** (*constants.AddressSpace*) – Address space into which move the data.
- **offset** (*int*) – Offset (in bytes) of the address or register from which to read.
- **length** (*int*) – Number of elements to transfer, where the data width of the elements to transfer is identical to the source data width.
- **data** (*Iterable[int]*) – Data to write to bus.
- **width** (*Union[Literal[8, 16, 32, 64], constants.DataWidth]*) – Number of bits to per element.
- **extended** (*bool, optional*) – Use 64 bits offset independent of the platform, by default False.

Returns Return value of the library call.

Return type *constants.StatusCode*

Raises ValueError – Raised if an invalid width is specified.

open(*access_mode: pyvisa.constants.AccessModes = <AccessModes.no_lock: 0>, open_timeout: int = 5000*) → None
Opens a session to the specified resource.

Parameters

- **access_mode** (*constants.AccessModes, optional*) – Specifies the mode by which the resource is to be accessed. Defaults to constants.AccessModes.no_lock.

- **open_timeout** (*int, optional*) – If the access_mode parameter requests a lock, then this parameter specifies the absolute time period (in milliseconds) that the resource waits to get unlocked before this operation returns an error. Defaults to 5000.

read_memory(*space: pyvisa.constants.AddressSpace*, *offset: int*, *width: pyvisa.constants.DataWidth*, *extended: bool = False*) → int

Read a value from the specified memory space and offset.

Parameters

- **space** (*constants.AddressSpace*) – Specifies the address space from which to read.
- **offset** (*int*) – Offset (in bytes) of the address or register from which to read.
- **width** (*Union[Literal[8, 16, 32, 64], constants.DataWidth]*) – Number of bits to read (8, 16, 32 or 64).
- **extended** (*bool, optional*) – Use 64 bits offset independent of the platform.

Returns **data** – Data read from memory

Return type int

Raises ValueError – Raised if an invalid width is specified.

classmethod register(*interface_type: pyvisa.constants.InterfaceType*, *resource_class: str*) → Callable[[Type[T]], Type[T]]

Create a decorator to register a class.

The class is associated to an interface type, resource class pair.

Parameters

- **interface_type** (constants.InterfaceType) – Interface type for which to register a wrapper class.
- **resource_class** (*str*) – Resource class for which to register a wrapper class.

Returns Decorator registering the class. Raises TypeError if some VISA attributes are missing on the registered class.

Return type Callable[[Type[T]], Type[T]]

resource_class

Resource class as defined by the canonical resource name.

Possible values are: INSTR, INTFC, SOCKET, RAW...

VISA Attribute VI_ATTR_RSRC_CLASS (3221159937)

resource_info

Get the extended information of this resource.

resource_manufacturer_name

Manufacturer name of the vendor that implemented the VISA library.

This attribute is not related to the device manufacturer attributes.

Note The value of this attribute is for display purposes only and not for programmatic decisions, as the value can differ between VISA implementations and/or revisions.

VISA Attribute VI_ATTR_RSRC_MANF_NAME (3221160308)

resource_name
Unique identifier for a resource compliant with the address structure. :VISA Attribute: VI_ATTR_RSRC_NAME (3221159938)

session
Resource session handle.

Raises `errors.InvalidSession` – Raised if session is closed.

set_visa_attribute(*name: pyvisa.constants.ResourceAttribute*, *state: Any*) → pyvisa.constants.StatusCode
Set the state of an attribute.

One should prefer the dedicated descriptor for often used attributes since those perform checks and automatic conversion on the value.

Parameters

- **name** (*constants.ResourceAttribute*) – Attribute for which the state is to be modified.
- **state** (*Any*) – The state of the attribute to be set for the specified object.

Returns Return value of the library call.

Return type *constants.StatusCode*

source_increment
Number of elements by which to increment the source offset after a transfer.

The default value of this attribute is 1 (that is, the source address will be incremented by 1 after each transfer), and the viMoveInXX() operations move from consecutive elements. If this attribute is set to 0, the viMoveInXX() operations will always read from the same element, essentially treating the source as a FIFO register.

VISA Attribute VI_ATTR_SRC_INCREMENT (1073676352)

Type int

Range 0 <= value <= 1

spec_version
Version of the VISA specification to which the implementation is compliant.

The format of the value has the upper 12 bits as the major number of the version, the next lower 12 bits as the minor number of the version, and the lowest 8 bits as the sub-minor number of the version. The current VISA specification defines the value to be 00300000h.

VISA Attribute VI_ATTR_RSRC_SPEC_VERSION (1073676656)

Type int

Range 0 <= value <= 4294967295

timeout
Timeout in milliseconds for all resource I/O operations.

This value is used when accessing the device associated with the given session.

Special values:

- **immediate (VI_TMO_IMMEDIATE): 0** (for convenience, any value smaller than 1 is considered as 0)

- **infinite (`VI_TMO_INFINITE`): `float('+inf')`** (for convenience, None is considered as `float('+inf')`)

To set an **infinite** timeout, you can also use:

```
>>> del instrument.timeout
```

A timeout value of VI_TMO_IMMEDIATE means that operations should never wait for the device to respond. A timeout value of VI_TMO_INFINITE disables the timeout mechanism.

VISA Attribute VI_ATTR_TMO_VALUE (1073676314)

Type int

Range 0 <= value <= 4294967295

uninstall_handler(*event_type: pyvisa.constants.EventType, handler: Callable[[NewType.<locals>.new_type, pyvisa.constants.EventType, NewType.<locals>.new_type, Any], None], user_handle=None*) → None

Uninstalls handlers for events in this resource.

Parameters

- **event_type** (*constants.EventType*) – Logical event identifier.
- **handler** (*VISAHandler*) – Handler function to be uninstalled by a client application.
- **user_handle** (*Any*) – The user handle returned by install_handler.

unlock() → None

Relinquishes a lock for the specified resource.

visa_attributes_classes = {<class 'pyvisa.attributes.AttrVI_ATTR_WIN_BASE_ADDR'>, <cla

wait_on_event(*in_event_type: pyvisa.constants.EventType, timeout: int, capture_timeout: bool = False*) → pyvisa.resources.resource.WaitResponse

Waits for an occurrence of the specified event in this resource.

in_event_type [constants.EventType] Logical identifier of the event(s) to wait for.

timeout [int] Absolute time period in time units that the resource shall wait for a specified event to occur before returning the time elapsed error. The time unit is in milliseconds. None means waiting forever if necessary.

capture_timeout [bool, optional] When True will not produce a VisaIOError(VI_ERROR_TMO) but instead return a WaitResponse with timed_out=True.

Returns Object that contains event_type, context and ret value.

Return type WaitResponse

wrap_handler(*callable: Callable[[Resource, pyvisa.events.Event, Any], None]*) → Callable[[NewType.<locals>.new_type, pyvisa.constants.EventType, NewType.<locals>.new_type, Any], None]

Wrap an event handler to provide the signature expected by VISA.

The handler is expected to have the following signature: handler(resource: Resource, event: Event, user_handle: Any) -> None.

The wrapped handler should be used only to handle events on the resource used to wrap the handler.

write_memory(*space: pyvisa.constants.AddressSpace*, *offset: int*, *data: int*, *width: pyvisa.constants.DataWidth*, *extended: bool = False*) → pyvisa.constants.StatusCode

Write a value to the specified memory space and offset.

Parameters

- **space** (*constants.AddressSpace*) – Specifies the address space.
- **offset** (*int*) – Offset (in bytes) of the address or register from which to read.
- **data** (*int*) – Data to write to bus.
- **width** (*Union[Literal[8, 16, 32, 64], constants.DataWidth]*) – Number of bits to read.
- **extended** (*bool, optional*) – Use 64 bits offset independent of the platform, by default False.

Returns Return value of the library call.

Return type *constants.StatusCode*

Raises ValueError – Raised if an invalid width is specified.

class pyvisa.resources.**VXIInstrument**(*resource_manager: pyvisa.highlevel.ResourceManager*, *resource_name: str*)

Communicates with to devices of type VXI::VXI logical address[::INSTR]

More complex resource names can be specified with the following grammar: VXI[board]::VXI logical address[::INSTR]

Do not instantiate directly, use *pyvisa.highlevel.ResourceManager.open_resource()*.

allow_dma

Should I/O accesses use DMA (True) or Programmed I/O (False).

In some implementations, this attribute may have global effects even though it is documented to be a local attribute. Since this affects performance and not functionality, that behavior is acceptable.

VISA Attribute VI_ATTR_DMA_ALLOW_EN (1073676318)

Type bool

before_close() → None

Called just before closing an instrument.

clear() → None

Clear this resource.

close() → None

Closes the VISA session and marks the handle as invalid.

destination_increment

Number of elements by which to increment the destination offset after a transfer.

The default value of this attribute is 1 (that is, the destination address will be incremented by 1 after each transfer), and the viMoveOutXX() operations move into consecutive elements. If this attribute is set to 0, the viMoveOutXX() operations will always write to the same element, essentially treating the destination as a FIFO register.

VISA Attribute VI_ATTR_DEST_INCREMENT (1073676353)

Type int

Range 0 <= value <= 1

disable_event(*event_type: pyvisa.constants.EventType*, *mechanism: pyvisa.constants.EventMechanism*) → None
Disable notification for an event type(s) via the specified mechanism(s).

Parameters

- **event_type** (*constants.EventType*) – Logical event identifier.
- **mechanism** (*constants.EventMechanism*) – Specifies event handling mechanisms to be disabled.

discard_events(*event_type: pyvisa.constants.EventType*, *mechanism: pyvisa.constants.EventMechanism*) → None
Discards event occurrences for an event type and mechanism in this resource.

Parameters

- **event_type** (*constants.EventType*) – Logical event identifier.
- **mechanism** (*constants.EventMechanism*) – Specifies event handling mechanisms to be disabled.

enable_event(*event_type: pyvisa.constants.EventType*, *mechanism: pyvisa.constants.EventMechanism*, *context: None = None*) → None
Enable event occurrences for specified event types and mechanisms in this resource.

Parameters

- **event_type** (*constants.EventType*) – Logical event identifier.
- **mechanism** (*constants.EventMechanism*) – Specifies event handling mechanisms to be enabled
- **context** (*None*) – Not currently used, leave as None.

get_visa_attribute(*name: pyvisa.constants.ResourceAttribute*) → Any
Retrieves the state of an attribute in this resource.

One should prefer the dedicated descriptor for often used attributes since those perform checks and automatic conversion on the value.

Parameters **name** (*constants.ResourceAttribute*) – Resource attribute for which the state query is made.

Returns The state of the queried attribute for a specified resource.

Return type Any

ignore_warning(**warnings_constants*) → AbstractContextManager[T_co]
Ignoring warnings context manager for the current resource.

Parameters **warnings_constants** (constants.StatusCode) – Constants identifying the warnings to ignore.

implementation_version
Resource version that identifies the revisions or implementations of a resource.

This attribute value is defined by the individual manufacturer and increments with each new revision. The format of the value has the upper 12 bits as the major number of the version, the next lower 12 bits as the minor number of the version, and the lowest 8 bits as the sub-minor number of the version.

VISA Attribute VI_ATTR_RSRC_IMPL_VERSION (1073676291)

Type int

Range 0 <= value <= 4294967295

install_handler(*event_type: pyvisa.constants.EventType, handler: Callable[[NewType.<locals>.new_type, pyvisa.constants.EventType, NewType.<locals>.new_type, Any], None], user_handle=None*) → Any

Install handlers for event callbacks in this resource.

Parameters

- **event_type** (*constants.EventType*) – Logical event identifier.
- **handler** (*VISAHandler*) – Handler function to be installed by a client application.
- **user_handle** – A value specified by an application that can be used for identifying handlers uniquely for an event type. Depending on the backend they may be restriction on the possible values. Look at the backend *install_visa_handler* for more details.

Returns User handle in a format amenable to the backend. This is this representation of the handle that should be used when unistalling a handler.

Return type Any

interface_number

Board number for the given interface. :VISA Attribute: VI_ATTR_INTF_NUM (1073676662) :type: int :range: 0 <= value <= 65535

interface_type

Interface type of the given session. :VISA Attribute: VI_ATTR_INTF_TYPE (1073676657) :type: :class:pyvisa.constants.InterfaceType

io_protocol

IO protocol to use. See the attribute definition for more details.

is_4882_compliant

Whether the device is 488.2 compliant.

last_status

Last status code for this session.

lock(*timeout: Union[float, typing_extensions.Literal['default']][default]] = 'default', requested_key: Optional[str] = None*) → str

Establish a shared lock to the resource.

Parameters

- **timeout** (*Union[float, Literal["default"]], optional*) – Absolute time period (in milliseconds) that a resource waits to get unlocked by the locking session before returning an error. Defaults to "default" which means use self.timeout.
- **requested_key** (*Optional[str], optional*) – Access key used by another session with which you want your session to share a lock or None to generate a new shared access key.

Returns A new shared access key if requested_key is None, otherwise, same value as the requested_key

Return type str

lock_context(*timeout: Union[float, typing_extensions.Literal['default']][default]] = 'default', requested_key: Optional[str] = 'exclusive'*) → Iterator[Optional[str]]

A context that locks

Parameters

- **timeout** (*Union[float, Literal["default"]], optional*) – Absolute time period (in milliseconds) that a resource waits to get unlocked by the locking session before returning an error. Defaults to "default" which means use self.timeout.
- **requested_key** (*Optional[str], optional*) – When using default of 'exclusive' the lock is an exclusive lock. Otherwise it is the access key for the shared lock or None to generate a new shared access key.

Yields *Optional[str]* – The access_key if applicable.

lock_excl(*timeout: Union[float, typing_extensions.Literal['default']][default]] = 'default'*) → None
Establish an exclusive lock to the resource.

Parameters **timeout** (*Union[float, Literal["default"]], optional*) – Absolute time period (in milliseconds) that a resource waits to get unlocked by the locking session before returning an error. Defaults to "default" which means use self.timeout.

lock_state
Current locking state of the resource.

The resource can be unlocked, locked with an exclusive lock, or locked with a shared lock.

VISA Attribute VI_ATTR_RSRC_LOCK_STATE (1073676292)

Type :class:pyvisa.constants.AccessModes

manufacturer_id
Manufacturer identification number of the device.

manufacturer_name
Manufacturer name.

model_code
Model code for the device.

model_name
Model name of the device.

move_in(*space: pyvisa.constants.AddressSpace, offset: int, length: int, width: pyvisa.constants.DataWidth, extended: bool = False*) → List[int]
Move a block of data to local memory from the given address space and offset.

Corresponds to viMoveIn* functions of the VISA library.

Parameters

- **space** (*constants.AddressSpace*) – Address space from which to move the data.
- **offset** (*int*) – Offset (in bytes) of the address or register from which to read.
- **length** (*int*) – Number of elements to transfer, where the data width of the elements to transfer is identical to the source data width.
- **width** (*Union[Literal[8, 16, 32, 64], constants.DataWidth]*) – Number of bits to read per element.
- **extended** (*bool, optional*) – Use 64 bits offset independent of the platform, by default False.

Returns

- **data** (*List[int]*) – Data read from the bus

- **status_code** (*constants.StatusCode*) – Return value of the library call.

Raises ValueError – Raised if an invalid width is specified.

move_out(*space: pyvisa.constants.AddressSpace, offset: int, length: int, data: Iterable[int], width: pyvisa.constants.DataWidth, extended: bool = False*) → pyvisa.constants.StatusCode

Move a block of data from local memory to the given address space and offset.

Corresponds to viMoveOut* functions of the VISA library.

Parameters

- **space** (*constants.AddressSpace*) – Address space into which move the data.
- **offset** (*int*) – Offset (in bytes) of the address or register from which to read.
- **length** (*int*) – Number of elements to transfer, where the data width of the elements to transfer is identical to the source data width.
- **data** (*Iterable[int]*) – Data to write to bus.
- **width** (*Union[Literal[8, 16, 32, 64], constants.DataWidth]*) – Number of bits to per element.
- **extended** (*bool, optional*) – Use 64 bits offset independent of the platform, by default False.

Returns Return value of the library call.

Return type *constants.StatusCode*

Raises ValueError – Raised if an invalid width is specified.

open(*access_mode: pyvisa.constants.AccessModes = <AccessModes.no_lock: 0>, open_timeout: int = 5000*) → None

Opens a session to the specified resource.

Parameters

- **access_mode** (constants.AccessModes, *optional*) – Specifies the mode by which the resource is to be accessed. Defaults to constants.AccessModes.no_lock.
- **open_timeout** (*int, optional*) – If the access_mode parameter requests a lock, then this parameter specifies the absolute time period (in milliseconds) that the resource waits to get unlocked before this operation returns an error. Defaults to 5000.

read_memory(*space: pyvisa.constants.AddressSpace, offset: int, width: pyvisa.constants.DataWidth, extended: bool = False*) → int

Read a value from the specified memory space and offset.

Parameters

- **space** (*constants.AddressSpace*) – Specifies the address space from which to read.
- **offset** (*int*) – Offset (in bytes) of the address or register from which to read.
- **width** (*Union[Literal[8, 16, 32, 64], constants.DataWidth]*) – Number of bits to read (8, 16, 32 or 64).
- **extended** (*bool, optional*) – Use 64 bits offset independent of the platform.

Returns **data** – Data read from memory

Return type int

Raises ValueError – Raised if an invalid width is specified.

classmethod register(*interface_type: pyvisa.constants.InterfaceType*, *resource_class: str*) → Callable[[Type[T]], Type[T]]

Create a decorator to register a class.

The class is associated to an interface type, resource class pair.

Parameters

- **interface_type** (constants.InterfaceType) – Interface type for which to register a wrapper class.
- **resource_class** (*str*) – Resource class for which to register a wrapper class.

Returns Decorator registering the class. Raises TypeError if some VISA attributes are missing on the registered class.

Return type Callable[[Type[T]], Type[T]]

resource_class

Resource class as defined by the canonical resource name.

Possible values are: INSTR, INTFC, SOCKET, RAW...

VISA Attribute VI_ATTR_RSRC_CLASS (3221159937)

resource_info

Get the extended information of this resource.

resource_manufacturer_name

Manufacturer name of the vendor that implemented the VISA library.

This attribute is not related to the device manufacturer attributes.

Note The value of this attribute is for display purposes only and not for programmatic decisions, as the value can differ between VISA implementations and/or revisions.

VISA Attribute VI_ATTR_RSRC_MANF_NAME (3221160308)

resource_name

Unique identifier for a resource compliant with the address structure. :VISA Attribute: VI_ATTR_RSRC_NAME (3221159938)

send_end

Should END be asserted during the transfer of the last byte of the buffer.

session

Resource session handle.

Raises errors.InvalidSession – Raised if session is closed.

set_visa_attribute(*name: pyvisa.constants.ResourceAttribute*, *state: Any*) → pyvisa.constants.StatusCode

Set the state of an attribute.

One should prefer the dedicated descriptor for often used attributes since those perform checks and automatic conversion on the value.

Parameters

- **name** (*constants.ResourceAttribute*) – Attribute for which the state is to be modified.
- **state** (*Any*) – The state of the attribute to be set for the specified object.

Returns Return value of the library call.

Return type *constants.StatusCode*

source_increment

Number of elements by which to increment the source offset after a transfer.

The default value of this attribute is 1 (that is, the source address will be incremented by 1 after each transfer), and the viMoveInXX() operations move from consecutive elements. If this attribute is set to 0, the viMoveInXX() operations will always read from the same element, essentially treating the source as a FIFO register.

VISA Attribute VI_ATTR_SRC_INCREMENT (1073676352)

Type int

Range 0 <= value <= 1

spec_version

Version of the VISA specification to which the implementation is compliant.

The format of the value has the upper 12 bits as the major number of the version, the next lower 12 bits as the minor number of the version, and the lowest 8 bits as the sub-minor number of the version. The current VISA specification defines the value to be 00300000h.

VISA Attribute VI_ATTR_RSRC_SPEC_VERSION (1073676656)

Type int

Range 0 <= value <= 4294967295

timeout

Timeout in milliseconds for all resource I/O operations.

This value is used when accessing the device associated with the given session.

Special values:

- **immediate (`VI_TMO_IMMEDIATE`): 0** (for convenience, any value smaller than 1 is considered as 0)
- **infinite (`VI_TMO_INFINITE`): `float('+inf')`** (for convenience, None is considered as `float('+inf')`)

To set an **infinite** timeout, you can also use:

```
>>> del instrument.timeout
```

A timeout value of VI_TMO_IMMEDIATE means that operations should never wait for the device to respond. A timeout value of VI_TMO_INFINITE disables the timeout mechanism.

VISA Attribute VI_ATTR_TMO_VALUE (1073676314)

Type int

Range 0 <= value <= 4294967295

uninstall_handler(*event_type: pyvisa.constants.EventType, handler: Callable[[NewType.<locals>.new_type, pyvisa.constants.EventType, NewType.<locals>.new_type, Any], None], user_handle=None*) → None

Uninstalls handlers for events in this resource.

Parameters

- **event_type** (*constants.EventType*) – Logical event identifier.
- **handler** (*VISAHandler*) – Handler function to be uninstalled by a client application.
- **user_handle** (*Any*) – The user handle returned by install_handler.

unlock() → None
Relinquishes a lock for the specified resource.

visa_attributes_classes = {<class 'pyvisa.attributes.AttrVI_ATTR_INTF_INST_NAME'>, <cl

wait_on_event(*in_event_type: pyvisa.constants.EventType*, *timeout: int*, *capture_timeout: bool = False*) → pyvisa.resources.resource.WaitResponse
Waits for an occurrence of the specified event in this resource.

in_event_type [constants.EventType] Logical identifier of the event(s) to wait for.

timeout [int] Absolute time period in time units that the resource shall wait for a specified event to occur before returning the time elapsed error. The time unit is in milliseconds. None means waiting forever if necessary.

capture_timeout [bool, optional] When True will not produce a VisaIOError(VI_ERROR_TMO) but instead return a WaitResponse with timed_out=True.

Returns Object that contains event_type, context and ret value.

Return type WaitResponse

wrap_handler(*callable: Callable[[Resource, pyvisa.events.Event, Any], None]*) → Callable[[NewType.<locals>.new_type, pyvisa.constants.EventType, NewType.<locals>.new_type, Any], None]
Wrap an event handler to provide the signature expected by VISA.

The handler is expected to have the following signature: handler(resource: Resource, event: Event, user_handle: Any) -> None.

The wrapped handler should be used only to handle events on the resource used to wrap the handler.

write_memory(*space: pyvisa.constants.AddressSpace*, *offset: int*, *data: int*, *width: pyvisa.constants.DataWidth*, *extended: bool = False*) → pyvisa.constants.StatusCode
Write a value to the specified memory space and offset.

Parameters

- **space** (*constants.AddressSpace*) – Specifies the address space.
- **offset** (*int*) – Offset (in bytes) of the address or register from which to read.
- **data** (*int*) – Data to write to bus.
- **width** (*Union[Literal[8, 16, 32, 64], constants.DataWidth]*) – Number of bits to read.
- **extended** (*bool, optional*) – Use 64 bits offset independent of the platform, by default False.

Returns Return value of the library call.

Return type *constants.StatusCode*

Raises ValueError – Raised if an invalid width is specified.

class pyvisa.resources.**VXIMemory**(*resource_manager: pyvisa.highlevel.ResourceManager*, *resource_name: str*)

Communicates with to devices of type VXI[board]::MEMACC

More complex resource names can be specified with the following grammar: VXI[board]::MEMACC

Do not instantiate directly, use *pyvisa.highlevel.ResourceManager.open_resource()*.

allow_dma

Should I/O accesses use DMA (True) or Programmed I/O (False).

In some implementations, this attribute may have global effects even though it is documented to be a local attribute. Since this affects performance and not functionality, that behavior is acceptable.

VISA Attribute VI_ATTR_DMA_ALLOW_EN (1073676318)

Type bool

before_close() → None

Called just before closing an instrument.

clear() → None

Clear this resource.

close() → None

Closes the VISA session and marks the handle as invalid.

destination_increment

Number of elements by which to increment the destination offset after a transfer.

The default value of this attribute is 1 (that is, the destination address will be incremented by 1 after each transfer), and the viMoveOutXX() operations move into consecutive elements. If this attribute is set to 0, the viMoveOutXX() operations will always write to the same element, essentially treating the destination as a FIFO register.

VISA Attribute VI_ATTR_DEST_INCREMENT (1073676353)

Type int

Range 0 <= value <= 1

disable_event(*event_type: pyvisa.constants.EventType*, *mechanism: pyvisa.constants.EventMechanism*) → None

Disable notification for an event type(s) via the specified mechanism(s).

Parameters

- **event_type** (*constants.EventType*) – Logical event identifier.
- **mechanism** (*constants.EventMechanism*) – Specifies event handling mechanisms to be disabled.

discard_events(*event_type: pyvisa.constants.EventType*, *mechanism: pyvisa.constants.EventMechanism*) → None

Discards event occurrences for an event type and mechanism in this resource.

Parameters

- **event_type** (*constants.EventType*) – Logical event identifier.
- **mechanism** (*constants.EventMechanism*) – Specifies event handling mechanisms to be disabled.

enable_event(*event_type: pyvisa.constants.EventType*, *mechanism: pyvisa.constants.EventMechanism*, *context: None = None*) → None
Enable event occurrences for specified event types and mechanisms in this resource.

Parameters

- **event_type** (*constants.EventType*) – Logical event identifier.
- **mechanism** (*constants.EventMechanism*) – Specifies event handling mechanisms to be enabled
- **context** (*None*) – Not currently used, leave as None.

get_visa_attribute(*name: pyvisa.constants.ResourceAttribute*) → Any
Retrieves the state of an attribute in this resource.

One should prefer the dedicated descriptor for often used attributes since those perform checks and automatic conversion on the value.

Parameters **name** (*constants.ResourceAttribute*) – Resource attribute for which the state query is made.

Returns The state of the queried attribute for a specified resource.

Return type Any

ignore_warning(**warnings_constants*) → AbstractContextManager[T_co]
Ignoring warnings context manager for the current resource.

Parameters **warnings_constants** (constants.StatusCode) – Constants identifying the warnings to ignore.

implementation_version
Resource version that identifies the revisions or implementations of a resource.

This attribute value is defined by the individual manufacturer and increments with each new revision. The format of the value has the upper 12 bits as the major number of the version, the next lower 12 bits as the minor number of the version, and the lowest 8 bits as the sub-minor number of the version.

VISA Attribute VI_ATTR_RSRC_IMPL_VERSION (1073676291)

Type int

Range 0 <= value <= 4294967295

install_handler(*event_type: pyvisa.constants.EventType, handler: Callable[[NewType.<locals>.new_type, pyvisa.constants.EventType, NewType.<locals>.new_type, Any], None], user_handle=None*) → Any
Install handlers for event callbacks in this resource.

Parameters

- **event_type** (*constants.EventType*) – Logical event identifier.
- **handler** (*VISAHandler*) – Handler function to be installed by a client application.
- **user_handle** – A value specified by an application that can be used for identifying handlers uniquely for an event type. Depending on the backend they may be restriction on the possible values. Look at the backend *install_visa_handler* for more details.

Returns User handle in a format amenable to the backend. This is this representation of the handle that should be used when unistalling a handler.

Return type Any

interface_number
Board number for the given interface. :VISA Attribute: VI_ATTR_INTF_NUM (1073676662) :type: int :range: 0 <= value <= 65535

interface_type
Interface type of the given session. :VISA Attribute: VI_ATTR_INTF_TYPE (1073676657) :type: :class:pyvisa.constants.InterfaceType

last_status
Last status code for this session.

lock(*timeout: Union[float, typing_extensions.Literal['default']][default]] = 'default', requested_key: Optional[str] = None*) → str
Establish a shared lock to the resource.

Parameters

- **timeout** (*Union[float, Literal["default"]], optional*) – Absolute time period (in milliseconds) that a resource waits to get unlocked by the locking session before returning an error. Defaults to "default" which means use self.timeout.
- **requested_key** (*Optional[str], optional*) – Access key used by another session with which you want your session to share a lock or None to generate a new shared access key.

Returns A new shared access key if requested_key is None, otherwise, same value as the requested_key

Return type str

lock_context(*timeout: Union[float, typing_extensions.Literal['default']][default]] = 'default', requested_key: Optional[str] = 'exclusive'*) → Iterator[Optional[str]]
A context that locks

Parameters

- **timeout** (*Union[float, Literal["default"]], optional*) – Absolute time period (in milliseconds) that a resource waits to get unlocked by the locking session before returning an error. Defaults to "default" which means use self.timeout.
- **requested_key** (*Optional[str], optional*) – When using default of 'exclusive' the lock is an exclusive lock. Otherwise it is the access key for the shared lock or None to generate a new shared access key.

Yields *Optional[str]* – The access_key if applicable.

lock_excl(*timeout: Union[float, typing_extensions.Literal['default']][default]] = 'default'*) → None
Establish an exclusive lock to the resource.

Parameters **timeout** (*Union[float, Literal["default"]], optional*) – Absolute time period (in milliseconds) that a resource waits to get unlocked by the locking session before returning an error. Defaults to "default" which means use self.timeout.

lock_state
Current locking state of the resource.

The resource can be unlocked, locked with an exclusive lock, or locked with a shared lock.

VISA Attribute VI_ATTR_RSRC_LOCK_STATE (1073676292)

Type :class:pyvisa.constants.AccessModes

move_in(*space: pyvisa.constants.AddressSpace, offset: int, length: int, width: pyvisa.constants.DataWidth, extended: bool = False*) → List[int]
Move a block of data to local memory from the given address space and offset.

Corresponds to viMoveIn* functions of the VISA library.

Parameters

- **space** (*constants.AddressSpace*) – Address space from which to move the data.
- **offset** (*int*) – Offset (in bytes) of the address or register from which to read.
- **length** (*int*) – Number of elements to transfer, where the data width of the elements to transfer is identical to the source data width.
- **width** (*Union[Literal[8, 16, 32, 64], constants.DataWidth]*) – Number of bits to read per element.
- **extended** (*bool, optional*) – Use 64 bits offset independent of the platform, by default False.

Returns

- **data** (*List[int]*) – Data read from the bus
- **status_code** (*constants.StatusCode*) – Return value of the library call.

Raises ValueError – Raised if an invalid width is specified.

move_out(*space: pyvisa.constants.AddressSpace, offset: int, length: int, data: Iterable[int], width: pyvisa.constants.DataWidth, extended: bool = False*) → pyvisa.constants.StatusCode
Move a block of data from local memory to the given address space and offset.

Corresponds to viMoveOut* functions of the VISA library.

Parameters

- **space** (*constants.AddressSpace*) – Address space into which move the data.
- **offset** (*int*) – Offset (in bytes) of the address or register from which to read.
- **length** (*int*) – Number of elements to transfer, where the data width of the elements to transfer is identical to the source data width.
- **data** (*Iterable[int]*) – Data to write to bus.
- **width** (*Union[Literal[8, 16, 32, 64], constants.DataWidth]*) – Number of bits to per element.
- **extended** (*bool, optional*) – Use 64 bits offset independent of the platform, by default False.

Returns Return value of the library call.

Return type *constants.StatusCode*

Raises ValueError – Raised if an invalid width is specified.

open(*access_mode: pyvisa.constants.AccessModes = <AccessModes.no_lock: 0>, open_timeout: int = 5000*) → None
Opens a session to the specified resource.

Parameters

- **access_mode** (*constants.AccessModes, optional*) – Specifies the mode by which the resource is to be accessed. Defaults to constants.AccessModes.no_lock.

- **open_timeout** (*int, optional*) – If the access_mode parameter requests a lock, then this parameter specifies the absolute time period (in milliseconds) that the resource waits to get unlocked before this operation returns an error. Defaults to 5000.

read_memory(*space: pyvisa.constants.AddressSpace*, *offset: int*, *width: pyvisa.constants.DataWidth*, *extended: bool = False*) → int

Read a value from the specified memory space and offset.

Parameters

- **space** (*constants.AddressSpace*) – Specifies the address space from which to read.
- **offset** (*int*) – Offset (in bytes) of the address or register from which to read.
- **width** (*Union[Literal[8, 16, 32, 64], constants.DataWidth]*) – Number of bits to read (8, 16, 32 or 64).
- **extended** (*bool, optional*) – Use 64 bits offset independent of the platform.

Returns **data** – Data read from memory

Return type int

Raises ValueError – Raised if an invalid width is specified.

classmethod register(*interface_type: pyvisa.constants.InterfaceType*, *resource_class: str*) → Callable[[Type[T]], Type[T]]

Create a decorator to register a class.

The class is associated to an interface type, resource class pair.

Parameters

- **interface_type** (constants.InterfaceType) – Interface type for which to register a wrapper class.
- **resource_class** (*str*) – Resource class for which to register a wrapper class.

Returns Decorator registering the class. Raises TypeError if some VISA attributes are missing on the registered class.

Return type Callable[[Type[T]], Type[T]]

resource_class

Resource class as defined by the canonical resource name.

Possible values are: INSTR, INTFC, SOCKET, RAW...

VISA Attribute VI_ATTR_RSRC_CLASS (3221159937)

resource_info

Get the extended information of this resource.

resource_manufacturer_name

Manufacturer name of the vendor that implemented the VISA library.

This attribute is not related to the device manufacturer attributes.

Note The value of this attribute is for display purposes only and not for programmatic decisions, as the value can differ between VISA implementations and/or revisions.

VISA Attribute VI_ATTR_RSRC_MANF_NAME (3221160308)

resource_name
Unique identifier for a resource compliant with the address structure. :VISA Attribute: VI_ATTR_RSRC_NAME (3221159938)

session
Resource session handle.

Raises `errors.InvalidSession` – Raised if session is closed.

set_visa_attribute(*name: pyvisa.constants.ResourceAttribute*, *state: Any*) → pyvisa.constants.StatusCode
Set the state of an attribute.

One should prefer the dedicated descriptor for often used attributes since those perform checks and automatic conversion on the value.

Parameters

- **name** (*constants.ResourceAttribute*) – Attribute for which the state is to be modified.
- **state** (*Any*) – The state of the attribute to be set for the specified object.

Returns Return value of the library call.

Return type *constants.StatusCode*

source_increment
Number of elements by which to increment the source offset after a transfer.

The default value of this attribute is 1 (that is, the source address will be incremented by 1 after each transfer), and the viMoveInXX() operations move from consecutive elements. If this attribute is set to 0, the viMoveInXX() operations will always read from the same element, essentially treating the source as a FIFO register.

VISA Attribute VI_ATTR_SRC_INCREMENT (1073676352)

Type int

Range 0 <= value <= 1

spec_version
Version of the VISA specification to which the implementation is compliant.

The format of the value has the upper 12 bits as the major number of the version, the next lower 12 bits as the minor number of the version, and the lowest 8 bits as the sub-minor number of the version. The current VISA specification defines the value to be 00300000h.

VISA Attribute VI_ATTR_RSRC_SPEC_VERSION (1073676656)

Type int

Range 0 <= value <= 4294967295

timeout
Timeout in milliseconds for all resource I/O operations.

This value is used when accessing the device associated with the given session.

Special values:

- **immediate (VI_TMO_IMMEDIATE): 0** (for convenience, any value smaller than 1 is considered as 0)

- **infinite (`VI_TMO_INFINITE`): `float('+inf')`** (for convenience, None is considered as `float('+inf')`)

To set an **infinite** timeout, you can also use:

```
>>> del instrument.timeout
```

A timeout value of VI_TMO_IMMEDIATE means that operations should never wait for the device to respond. A timeout value of VI_TMO_INFINITE disables the timeout mechanism.

VISA Attribute VI_ATTR_TMO_VALUE (1073676314)

Type int

Range 0 <= value <= 4294967295

uninstall_handler(*event_type: pyvisa.constants.EventType, handler: Callable[[NewType.<locals>.new_type, pyvisa.constants.EventType, NewType.<locals>.new_type, Any], None], user_handle=None*) → None

Uninstalls handlers for events in this resource.

Parameters

- **event_type** (*constants.EventType*) – Logical event identifier.
- **handler** (*VISAHandler*) – Handler function to be uninstalled by a client application.
- **user_handle** (*Any*) – The user handle returned by install_handler.

unlock() → None

Relinquishes a lock for the specified resource.

visa_attributes_classes = {<class 'pyvisa.attributes.AttrVI_ATTR_INTF_INST_NAME'>, <cl

wait_on_event(*in_event_type: pyvisa.constants.EventType, timeout: int, capture_timeout: bool = False*) → pyvisa.resources.resource.WaitResponse

Waits for an occurrence of the specified event in this resource.

in_event_type [constants.EventType] Logical identifier of the event(s) to wait for.

timeout [int] Absolute time period in time units that the resource shall wait for a specified event to occur before returning the time elapsed error. The time unit is in milliseconds. None means waiting forever if necessary.

capture_timeout [bool, optional] When True will not produce a VisaIOError(VI_ERROR_TMO) but instead return a WaitResponse with timed_out=True.

Returns Object that contains event_type, context and ret value.

Return type WaitResponse

wrap_handler(*callable: Callable[[Resource, pyvisa.events.Event, Any], None]*) → Callable[[NewType.<locals>.new_type, pyvisa.constants.EventType, NewType.<locals>.new_type, Any], None]

Wrap an event handler to provide the signature expected by VISA.

The handler is expected to have the following signature: handler(resource: Resource, event: Event, user_handle: Any) -> None.

The wrapped handler should be used only to handle events on the resource used to wrap the handler.

write_memory(*space: pyvisa.constants.AddressSpace*, *offset: int*, *data: int*, *width: pyvisa.constants.DataWidth*, *extended: bool = False*) → pyvisa.constants.StatusCode
Write a value to the specified memory space and offset.

Parameters

- **space** (*constants.AddressSpace*) – Specifies the address space.
- **offset** (*int*) – Offset (in bytes) of the address or register from which to read.
- **data** (*int*) – Data to write to bus.
- **width** (*Union[Literal[8, 16, 32, 64], constants.DataWidth]*) – Number of bits to read.
- **extended** (*bool, optional*) – Use 64 bits offset independent of the platform, by default False.

Returns Return value of the library call.

Return type *constants.StatusCode*

Raises ValueError – Raised if an invalid width is specified.

class pyvisa.resources.**VXIBackplane**(*resource_manager: pyvisa.highlevel.ResourceManager*, *resource_name: str*)
Communicates with to devices of type VXI::BACKPLANE

More complex resource names can be specified with the following grammar: VXI[board][::VXI logical address]::BACKPLANE

Do not instantiate directly, use *pyvisa.highlevel.ResourceManager.open_resource()*.

before_close() → None
Called just before closing an instrument.

clear() → None
Clear this resource.

close() → None
Closes the VISA session and marks the handle as invalid.

disable_event(*event_type: pyvisa.constants.EventType*, *mechanism: pyvisa.constants.EventMechanism*) → None
Disable notification for an event type(s) via the specified mechanism(s).

Parameters

- **event_type** (*constants.EventType*) – Logical event identifier.
- **mechanism** (*constants.EventMechanism*) – Specifies event handling mechanisms to be disabled.

discard_events(*event_type: pyvisa.constants.EventType*, *mechanism: pyvisa.constants.EventMechanism*) → None
Discards event occurrences for an event type and mechanism in this resource.

Parameters

- **event_type** (*constants.EventType*) – Logical event identifier.
- **mechanism** (*constants.EventMechanism*) – Specifies event handling mechanisms to be disabled.

enable_event(*event_type: pyvisa.constants.EventType*, *mechanism: pyvisa.constants.EventMechanism*, *context: None = None*) → None
Enable event occurrences for specified event types and mechanisms in this resource.

Parameters

- **event_type** (*constants.EventType*) – Logical event identifier.
- **mechanism** (*constants.EventMechanism*) – Specifies event handling mechanisms to be enabled
- **context** (*None*) – Not currently used, leave as None.

get_visa_attribute(*name: pyvisa.constants.ResourceAttribute*) → Any
Retrieves the state of an attribute in this resource.

One should prefer the dedicated descriptor for often used attributes since those perform checks and automatic conversion on the value.

Parameters **name** (*constants.ResourceAttribute*) – Resource attribute for which the state query is made.

Returns The state of the queried attribute for a specified resource.

Return type Any

ignore_warning(**warnings_constants*) → AbstractContextManager[T_co]
Ignoring warnings context manager for the current resource.

Parameters **warnings_constants** (constants.StatusCode) – Constants identifying the warnings to ignore.

implementation_version
Resource version that identifies the revisions or implementations of a resource.

This attribute value is defined by the individual manufacturer and increments with each new revision. The format of the value has the upper 12 bits as the major number of the version, the next lower 12 bits as the minor number of the version, and the lowest 8 bits as the sub-minor number of the version.

VISA Attribute VI_ATTR_RSRC_IMPL_VERSION (1073676291)

Type int

Range 0 <= value <= 4294967295

install_handler(*event_type: pyvisa.constants.EventType, handler: Callable[[NewType.<locals>.new_type, pyvisa.constants.EventType, NewType.<locals>.new_type, Any], None], user_handle=None*) → Any
Install handlers for event callbacks in this resource.

Parameters

- **event_type** (*constants.EventType*) – Logical event identifier.
- **handler** (*VISAHandler*) – Handler function to be installed by a client application.
- **user_handle** – A value specified by an application that can be used for identifying handlers uniquely for an event type. Depending on the backend they may be restriction on the possible values. Look at the backend *install_visa_handler* for more details.

Returns User handle in a format amenable to the backend. This is this representation of the handle that should be used when unistalling a handler.

Return type Any

interface_number
Board number for the given interface. :VISA Attribute: VI_ATTR_INTF_NUM (1073676662) :type: int :range: 0 <= value <= 65535

interface_type
Interface type of the given session. :VISA Attribute: VI_ATTR_INTF_TYPE (1073676657) :type: :class:pyvisa.constants.InterfaceType

last_status
Last status code for this session.

lock(*timeout: Union[float, typing_extensions.Literal['default']][default]] = 'default', requested_key: Optional[str] = None*) → str
Establish a shared lock to the resource.

Parameters

- **timeout** (*Union[float, Literal["default"]], optional*) – Absolute time period (in milliseconds) that a resource waits to get unlocked by the locking session before returning an error. Defaults to "default" which means use self.timeout.
- **requested_key** (*Optional[str], optional*) – Access key used by another session with which you want your session to share a lock or None to generate a new shared access key.

Returns A new shared access key if requested_key is None, otherwise, same value as the requested_key

Return type str

lock_context(*timeout: Union[float, typing_extensions.Literal['default']][default]] = 'default', requested_key: Optional[str] = 'exclusive'*) → Iterator[Optional[str]]
A context that locks

Parameters

- **timeout** (*Union[float, Literal["default"]], optional*) – Absolute time period (in milliseconds) that a resource waits to get unlocked by the locking session before returning an error. Defaults to "default" which means use self.timeout.
- **requested_key** (*Optional[str], optional*) – When using default of 'exclusive' the lock is an exclusive lock. Otherwise it is the access key for the shared lock or None to generate a new shared access key.

Yields *Optional[str]* – The access_key if applicable.

lock_excl(*timeout: Union[float, typing_extensions.Literal['default']][default]] = 'default'*) → None
Establish an exclusive lock to the resource.

Parameters **timeout** (*Union[float, Literal["default"]], optional*) – Absolute time period (in milliseconds) that a resource waits to get unlocked by the locking session before returning an error. Defaults to "default" which means use self.timeout.

lock_state
Current locking state of the resource.

The resource can be unlocked, locked with an exclusive lock, or locked with a shared lock.

VISA Attribute VI_ATTR_RSRC_LOCK_STATE (1073676292)

Type :class:pyvisa.constants.AccessModes

open(*access_mode: pyvisa.constants.AccessModes = <AccessModes.no_lock: 0>, open_timeout: int = 5000*) → None
Opens a session to the specified resource.

Parameters

- **access_mode** (constants.AccessModes, *optional*) – Specifies the mode by which the resource is to be accessed. Defaults to constants.AccessModes.no_lock.
- **open_timeout** (*int*, *optional*) – If the access_mode parameter requests a lock, then this parameter specifies the absolute time period (in milliseconds) that the resource waits to get unlocked before this operation returns an error. Defaults to 5000.

classmethod register(*interface_type: pyvisa.constants.InterfaceType*, *resource_class: str*) → Callable[[Type[T]], Type[T]]

Create a decorator to register a class.

The class is associated to an interface type, resource class pair.

Parameters

- **interface_type** (constants.InterfaceType) – Interface type for which to register a wrapper class.
- **resource_class** (*str*) – Resource class for which to register a wrapper class.

Returns Decorator registering the class. Raises TypeError if some VISA attributes are missing on the registered class.

Return type Callable[[Type[T]], Type[T]]

resource_class

Resource class as defined by the canonical resource name.

Possible values are: INSTR, INTFC, SOCKET, RAW...

VISA Attribute VI_ATTR_RSRC_CLASS (3221159937)

resource_info

Get the extended information of this resource.

resource_manufacturer_name

Manufacturer name of the vendor that implemented the VISA library.

This attribute is not related to the device manufacturer attributes.

Note The value of this attribute is for display purposes only and not for programmatic decisions, as the value can differ between VISA implementations and/or revisions.

VISA Attribute VI_ATTR_RSRC_MANF_NAME (3221160308)

resource_name

Unique identifier for a resource compliant with the address structure. :VISA Attribute: VI_ATTR_RSRC_NAME (3221159938)

session

Resource session handle.

Raises errors.InvalidSession – Raised if session is closed.

set_visa_attribute(*name: pyvisa.constants.ResourceAttribute*, *state: Any*) → pyvisa.constants.StatusCode

Set the state of an attribute.

One should prefer the dedicated descriptor for often used attributes since those perform checks and automatic conversion on the value.

Parameters

- **name** (*constants.ResourceAttribute*) – Attribute for which the state is to be modified.
- **state** (*Any*) – The state of the attribute to be set for the specified object.

Returns Return value of the library call.

Return type *constants.StatusCode*

spec_version
Version of the VISA specification to which the implementation is compliant.

The format of the value has the upper 12 bits as the major number of the version, the next lower 12 bits as the minor number of the version, and the lowest 8 bits as the sub-minor number of the version. The current VISA specification defines the value to be 00300000h.

VISA Attribute VI_ATTR_RSRC_SPEC_VERSION (1073676656)

Type int

Range 0 <= value <= 4294967295

timeout
Timeout in milliseconds for all resource I/O operations.

This value is used when accessing the device associated with the given session.

Special values:

- **immediate (VI_TMO_IMMEDIATE): 0** (for convenience, any value smaller than 1 is considered as 0)
- **infinite (VI_TMO_INFINITE): float('+inf')** (for convenience, None is considered as `float('+inf')`)

To set an **infinite** timeout, you can also use:

```
>>> del instrument.timeout
```

A timeout value of VI_TMO_IMMEDIATE means that operations should never wait for the device to respond. A timeout value of VI_TMO_INFINITE disables the timeout mechanism.

VISA Attribute VI_ATTR_TMO_VALUE (1073676314)

Type int

Range 0 <= value <= 4294967295

uninstall_handler(*event_type: pyvisa.constants.EventType, handler: Callable[[NewType.<locals>.new_type, pyvisa.constants.EventType, NewType.<locals>.new_type, Any], None], user_handle=None*) → None
Uninstalls handlers for events in this resource.

Parameters

- **event_type** (*constants.EventType*) – Logical event identifier.
- **handler** (*VISAHandler*) – Handler function to be uninstalled by a client application.
- **user_handle** (*Any*) – The user handle returned by install_handler.

unlock() → None
Relinquishes a lock for the specified resource.

visa_attributes_classes = {<class 'pyvisa.attributes.AttrVI_ATTR_INTF_INST_NAME'>, <cl

wait_on_event(*in_event_type: pyvisa.constants.EventType*, *timeout: int*, *capture_timeout: bool = False*) → pyvisa.resources.resource.WaitResponse

Waits for an occurrence of the specified event in this resource.

in_event_type [constants.EventType] Logical identifier of the event(s) to wait for.

timeout [int] Absolute time period in time units that the resource shall wait for a specified event to occur before returning the time elapsed error. The time unit is in milliseconds. None means waiting forever if necessary.

capture_timeout [bool, optional] When True will not produce a VisaIOError(VI_ERROR_TMO) but instead return a WaitResponse with timed_out=True.

Returns Object that contains event_type, context and ret value.

Return type WaitResponse

wrap_handler(*callable: Callable[[Resource, pyvisa.events.Event, Any], None]*) → Callable[[NewType.<locals>.new_type, pyvisa.constants.EventType, NewType.<locals>.new_type, Any], None]

Wrap an event handler to provide the signature expected by VISA.

The handler is expected to have the following signature: handler(resource: Resource, event: Event, user_handle: Any) -> None.

The wrapped handler should be used only to handle events on the resource used to wrap the handler.

1.4.4 Constants module

Provides user-friendly naming to values used in different functions.

class pyvisa.constants.**AccessModes**

Whether and how to lock a resource when opening a connection.

exclusive_lock = 1

Obtains a exclusive lock on the VISA resource.

no_lock = 0

Does not obtain any lock on the VISA resource.

shared_lock = 2

Obtains a lock on the VISA resouce which may be shared between multiple VISA sessions.

class pyvisa.constants.**StopBits**

The number of stop bits that indicate the end of a frame on a serial resource.

Used only for ASRL resources.

one = 10

one_and_a_half = 15

two = 20

class pyvisa.constants.**Parity**

Parity type to use with every frame transmitted and received on a serial session.

Used only for ASRL resources.

even = 2

mark = 3

none = 0

odd = 1

space = 4

class pyvisa.constants.**SerialTermination**
The available methods for terminating a serial transfer.

last_bit = 1
The transfer occurs with the last bit not set until the last character is sent.

none = 0
The transfer terminates when all requested data is transferred or when an error occurs.

termination_break = 3
The write transmits a break after all the characters for the write are sent.

termination_char = 2
The transfer terminate by searching for "/" appending the termination character.

class pyvisa.constants.**InterfaceType**
The hardware interface.

asrl = 4
Serial devices connected to either an RS-232 or RS-485 controller.

firewire = 9
Firewire device.

gpib = 1
GPIB Interface.

gpib_vxi = 3
GPIB VXI (VME eXtensions for Instrumentation).

pxi = 5
PXI device.

rio = 8
Rio device.

rsnrp = 33024
Rohde and Schwarz Device via Passport

tcpip = 6
TCPIP device.

unknown = -1

usb = 7
Universal Serial Bus (USB) hardware bus.

vxi = 2
VXI (VME eXtensions for Instrumentation), VME, MXI (Multisystem eXtension Interface).

class pyvisa.constants.**AddressState**
State of a GPIB resource.

Corresponds to the Attribute.GPIB_address_state attribute

listenr = 2
The resource is addressed to listen

talker = 1
The resource is addressed to talk

unaddressed = 0
The resource is unadressed

class pyvisa.constants.**IOProtocol**
IO protocol used for communication.

See attributes.AttrVI_ATTR_IO_PROT for more details.

fdc = 2
Fast data channel (FDC) protocol for VXI

hs488 = 3
High speed 488 transfer for GPIB

normal = 1

protocol4882_strs = 4
488 style transfer for serial

usbtmc_vendor = 5
Test measurement class vendor specific for USB

class pyvisa.constants.**LineState**
State of a hardware line or signal.

The line for which the state can be queried are: - ASRC resource: BREAK, CTS, DCD, DSR, DTR, RI, RTS signals - GPIB resources: ATN, NDAC, REN, SRQ lines - VXI BACKPLANE: VXI/VME SYSFAIL backplane line

Search for LineState in attributes.py for more details.

asserted = 1
The line/signal is currently asserted

unasserted = 0
The line/signal is currently deasserted

unknown = -1
The state of the line/signal is unknown

class pyvisa.constants.**StatusCode**
Status codes that VISA driver-level operations can return.

error_abort = -1073807312
The operation was aborted.

error_allocation = -1073807300
Insufficient system resources to perform necessary memory allocation.

error_attribute_read_only = -1073807329
The specified attribute is read-only.

error_bus_error = -1073807304
Bus error occurred during transfer.

error_closing_failed = -1073807338
Unable to deallocate the previously allocated data structures corresponding to this session or object reference.

error_connection_lost = -1073807194
The connection for the specified session has been lost.

error_file_access = -1073807199
An error occurred while trying to open the specified file. Possible causes include an invalid path or lack of access rights.

error_file_i_o = -1073807198
An error occurred while performing I/O on the specified file.

error_handler_not_installed = -1073807320
A handler is not currently installed for the specified event.

error_in_progress = -1073807303
Unable to queue the asynchronous operation because there is already an operation in progress.

error_input_protocol_violation = -1073807305
Device reported an input protocol error during transfer.

error_interface_number_not_configured = -1073807195
The interface type is valid but the specified interface number is not configured.

error_interrupt_pending = -1073807256
An interrupt is still pending from a previous call.

error_invalid_access_key = -1073807327
The access key to the resource associated with this session is invalid.

error_invalid_access_mode = -1073807341
Invalid access mode.

error_invalid_address_space = -1073807282
Invalid address space specified.

error_invalid_context = -1073807318
Specified event context is invalid.

error_invalid_degree = -1073807333
Specified degree is invalid.

error_invalid_event = -1073807322
Specified event type is not supported by the resource.

error_invalid_expression = -1073807344
Invalid expression specified for search.

error_invalid_format = -1073807297
A format specifier in the format string is invalid.

error_invalid_handler_reference = -1073807319
The specified handler reference is invalid.

error_invalid_job_i_d = -1073807332
Specified job identifier is invalid.

error_invalid_length = -1073807229
Invalid length specified.

error_invalid_line = -1073807200
The value specified by the line parameter is invalid.

error_invalid_lock_type = -1073807328
The specified type of lock is not supported by this resource.

error_invalid_mask = -1073807299
Invalid buffer mask specified.

error_invalid_mechanism = -1073807321
Invalid mechanism specified.

error_invalid_mode = -1073807215
The specified mode is invalid.

error_invalid_object = -1073807346
The specified session or object reference is invalid.

error_invalid_offset = -1073807279
Invalid offset specified.

error_invalid_parameter = -1073807240
The value of an unknown parameter is invalid.

error_invalid_protocol = -1073807239
The protocol specified is invalid.

error_invalid_resource_name = -1073807342
Invalid resource reference specified. Parsing error.

error_invalid_setup = -1073807302
Unable to start operation because setup is invalid due to inconsistent state of properties.

error_invalid_size = -1073807237
Invalid size of window specified.

error_invalid_width = -1073807278
Invalid source or destination width specified.

error_io = -1073807298
Could not perform operation because of I/O error.

error_library_not_found = -1073807202
A code library required by VISA could not be located or loaded.

error_line_in_use = -1073807294
The specified trigger line is currently in use.

error_machine_not_available = -1073807193
The remote machine does not exist or is not accepting any connections.

error_memory_not_shared = -1073807203
The device does not export any memory.

error_no_listeners = -1073807265
No listeners condition is detected (both NRFD and NDAC are deasserted).

error_no_permission = -1073807192
Access to the remote machine is denied.

error_nonimplemented_operation = -1073807231
The specified operation is unimplemented.

error_nonsupported_attribute = -1073807331
The specified attribute is not defined or supported by the referenced session, event, or find list.

error_nonsupported_attribute_state = -1073807330
The specified state of the attribute is not valid or is not supported as defined by the session, event, or find list.

error_nonsupported_format = -1073807295
A format specifier in the format string is not supported.

error_nonsupported_interrupt = -1073807201
The interface cannot generate an interrupt on the requested level or with the requested statusID value.

error_nonsupported_line = -1073807197
The specified trigger source line (trigSrc) or destination line (trigDest) is not supported by this VISA implementation, or the combination of lines is not a valid mapping.

error_nonsupported_mechanism = -1073807196
The specified mechanism is not supported for the specified event type.

error_nonsupported_mode = -1073807290
The specified mode is not supported by this VISA implementation.

error_nonsupported_offset = -1073807276
Specified offset is not accessible from this hardware.

error_nonsupported_offset_alignment = -1073807248
The specified offset is not properly aligned for the access width of the operation.

error_nonsupported_operation = -1073807257
The session or object reference does not support this operation.

error_nonsupported_varying_widths = -1073807275
Cannot support source and destination widths that are different.

error_nonsupported_width = -1073807242
Specified width is not supported by this hardware.

error_not_cic = -1073807264
The interface associated with this session is not currently the Controller-in-Charge.

error_not_enabled = -1073807313
The session must be enabled for events of the specified type in order to receive them.

error_not_system_controller = -1073807263
The interface associated with this session is not the system controller.

error_output_protocol_violation = -1073807306
Device reported an output protocol error during transfer.

error_queue_error = -1073807301
Unable to queue asynchronous operation.

error_queue_overflow = -1073807315
The event queue for the specified type has overflowed, usually due to not closing previous events.

error_raw_read_protocol_violation = -1073807307
Violation of raw read protocol occurred during transfer.

error_raw_write_protocol_violation = -1073807308
Violation of raw write protocol occurred during transfer.

error_resource_busy = -1073807246
The resource is valid, but VISA cannot currently access it.

error_resource_locked = -1073807345
Specified type of lock cannot be obtained or specified operation cannot be performed because the resource is locked.

error_resource_not_found = -1073807343
Insufficient location information, or the device or resource is not present in the system.

error_response_pending = -1073807271
A previous response is still pending, causing a multiple query error.

error_serial_framing = -1073807253
A framing error occurred during transfer.

error_serial_overrun = -1073807252
An overrun error occurred during transfer. A character was not read from the hardware before the next character arrived.

error_serial_parity = -1073807254
A parity error occurred during transfer.

error_session_not_locked = -1073807204
The current session did not have any lock on the resource.

error_srq_not_occurred = -1073807286
Service request has not been received for the session.

error_system_error = -1073807360
Unknown system error.

error_timeout = -1073807339
Timeout expired before operation completed.

error_trigger_not_mapped = -1073807250
The path from the trigger source line (trigSrc) to the destination line (trigDest) is not currently mapped.

error_user_buffer = -1073807247
A specified user buffer is not valid or cannot be accessed for the required size.

error_window_already_mapped = -1073807232
The specified session currently contains a mapped window.

error_window_not_mapped = -1073807273
The specified session is currently unmapped.

success = 0
Operation completed successfully.

success_device_not_present = 1073676413
Session opened successfully, but the device at the specified address is not responding.

success_event_already_disabled = 1073676291
Specified event is already disabled for at least one of the specified mechanisms.

success_event_already_enabled = 1073676290
Specified event is already enabled for at least one of the specified mechanisms.

success_max_count_read = 1073676294
The number of bytes read is equal to the input count.

success_nested_exclusive = 1073676442
Operation completed successfully, and this session has nested exclusive locks.

success_nested_shared = 1073676441
Operation completed successfully, and this session has nested shared locks.

success_no_more_handler_calls_in_chain = 1073676440
Event handled successfully. Do not invoke any other handlers on this session for this event.

success_queue_already_empty = 1073676292
Operation completed successfully, but the queue was already empty.

success_queue_not_empty = 1073676416
Wait terminated successfully on receipt of an event notification. There is still at least one more event occurrence of the requested type(s) available for this session.

success_synchronous = 1073676443
Asynchronous operation request was performed synchronously.

success_termination_character_read = 1073676293
The specified termination character was read.

success_trigger_already_mapped = 1073676414
The path from the trigger source line (trigSrc) to the destination line (trigDest) is already mapped.

warning_configuration_not_loaded = 1073676407
The specified configuration either does not exist or could not be loaded. The VISA-specified defaults are used.

warning_ext_function_not_implemented = 1073676457
The operation succeeded, but a lower level driver did not implement the extended functionality.

warning_nonsupported_attribute_state = 1073676420
Although the specified state of the attribute is valid, it is not supported by this resource implementation.

warning_nonsupported_buffer = 1073676424
The specified buffer is not supported.

warning_null_object = 1073676418
The specified object reference is uninitialized.

warning_queue_overflow = 1073676300
VISA received more event information of the specified type than the configured queue size could hold.

warning_unknown_status = 1073676421
The status code passed to the operation could not be interpreted.

Python Module Index

p

Index

C

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

W

X