PyVISA Documentation

Release 1.5

PyVISA Authors

August 23, 2014

Contents

1 General overview **3**

2 User guide **5**
- 2.1 Installation 5
- 2.2 Tutorial 6
- 2.3 Configuring PyVISA 8
- 2.4 Advanced 9
- 2.5 Instruments 10
- 2.6 Architecture 15

3 More information **17**
- 3.1 Migrating from PyVISA < 1.5 17
- 3.2 Contributing to PyVISA 20
- 3.3 Frequently asked questions 21
- 3.4 API 23

4 Legacy Modules **59**
- 4.1 About the legacy visa module 59
- 4.2 About the legacy vpp43 module 62
- 4.3 Legacy API 63

Python Module Index **93**

Warning: This documentation corresponds to PyVISA 1.5. If you are still using the old version, please update using following command:

```
pip install -U pyvisa
```

Just in case you need them, the old docs are here: http://pyvisa.readthedocs.org/en/1.4/

The PyVISA package enables you to control all kinds of measurement equipment through various busses (GPIB, RS232, USB) with Python programs. As an example, reading self-identification from a Keithley Multimeter with GPIB number 12 is as easy as three lines of Python code:

```
>>> import visa
>>> rm = visa.ResourceManager()
>>> rm.list_resources()
['ASRL1', 'ASRL2', 'GPIB::12']
>>> keithley = rm.get_instrument("GPIB::12")
>>> print(keithley.ask("*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). In order to achieve this, PyVISA relies on an external library file which is bundled with hardware and software of those vendors.

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 (GPIB, RS232, USB). 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 VISA (Virtual Instrument Software Architecture specification was defined in the middle of the 90ies. 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). However, you can use VISA with any language capable of calling functions in a shared library (*.dll*, *.so*, *.dylib*). PyVISA is Python wrapper for such shared library.

CHAPTER 2

User guide

2.1 Installation

PyVISA is a wrapper around the *National Instruments's VISA* library, which you need to download and install in order to use PyVISA (*getting_nivisa*).

PyVISA has no additional dependencies except Python itself. In runs on Python 2.6+ and 3.2+.

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

You can install it using pip:

```
$ pip install pyvisa
```

or using easy_install:

```
$ easy_install pyvisa
```

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

```
>>> import visa
>>> lib = visa.VisaLibrary()
```

If you encounter any problem, take a look at the *Frequently asked questions*.

2.1.1 Getting the code

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

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

Download the tarball:

```
$ curl -OL https://github.com/hgrecco/pyvisa/tarball/master
```

Or, download the zipball:

```
$ curl -OL https://github.com/hgrecco/pyvisa/zipball/master
```

Once you have a copy of the source, you can embed it in your Python package, or install it into your site-packages easily:

```
$ python setup.py install
```

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

2.2 Tutorial

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

2.2.1 An example

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

```
>>> import visa
>>> rm = visa.ResourceManager()
>>> my_instrument = rm.get_instrument('GPIB::14')
>>> my_instrument.write("*IDN?")
>>> print(my_instrument.read())
```

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

Afer importing *visa*, we create a *ResourceManager* object. If called without arguments, PyVISA will try to find the VISA shared for you. You can check, the location of the shared library used 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 PyVISA*.

Once that you havea *ResourceManager*, you can access any instrument. Every instrument is represented in the source by an object instance. In this case, I have a GPIB instrument with instrument number 14, so I create the instance (i.e. variable) called *my_instrument* accordingly with *"GPIB::14"* is the instrument's *resource name*. Notice that eventhough you have requeste an instrument, due to the resource name, *get_instrument* has given you an instance of *GpibInstrument* class (a subclass of the more generic instrument).

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

See section *VISA resource names* for a short explanation of that. Then, I send the message *"*IDN?"* to the device, which is the standard GPIB message for "what are you?" or – in some cases – "what's on your display at the moment?".

2.2.2 Listing connected instruments

The resource manager object allows you to list available resources:

```
>>> rm.list_resources()
['ASRL1', 'ASRL2']
```

or the most comprehensive *list_resources_info* which return a dict mapping resource name to a namedtuple containing information such as the interface type and the resource class.

2.2.3 Example for serial (RS232) device

There is no only RS232 device in my lab is an old Oxford ITC4 temperature controller, which is connected through COM2 with my computer. The following code prints its self-identification on the screen:

```
itc4 = rm.get_instrument("COM2")
itc4.write("V")
print(itc4.read())
```

Instead of separate write and read operations, you can do both with one *ask()* call. Thus, the above source code is equivalent to:

```
from visa import *

itc4 = instrument("COM2")
print(itc4.ask("V"))
```

It couldn't be simpler. See section *Serial devices* for further information about serial devices.

2.2.4 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 initialise the instrument:

```
>>> keithley = rm.get_instrument("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 initialisation 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; 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.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.ask("trace:data?")* however, then we'd get

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

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

```
>>> voltages = keithley.ask_for_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.ask("status:measurement?")
>>> keithley.write("trace:clear; feed:control next")
```

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

2.2.5 VISA resource names

If you use the function `get_instrument()`, 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.

2.3 Configuring PyVISA

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 *VisaLibrary* or *ResourceManager* constructor:

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

or:

```
>>> 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.

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:\Python27\share\pyvisa\.pyvisarc`.

2.4 Advanced

You can mix the high-level object-oriented approach described in this document with middle- and low-level VISA function calls (See *Architecture* for more information). By doing so, you have full control of your devices:

After you have instantiated the *ResourceManager*:

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

you can access corresponding the *VisaLibrary* instance under the *visalib*. attribute. The *VisaLibrary* object contains low-level functions as directly exposed by the foreign library, 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. 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.

You can recognize low an 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.

2.5 Instruments

class `Instrument` (*resource_name*[, ***keyw*])

represents an instrument, e.g. a measurement device. It is independent of a particular bus system, i.e. it may be a GPIB, serial, USB, or whatever instrument. However, it is not possible to perform bus-specific operations on instruments created by this class. For this, have a look at the specialised classes like `GpibInstrument` (section *Common properties of instrument variables*).

The parameter *resource_name* takes the same syntax as resource specifiers in VISA. Thus, it begins with the bus system followed by *"::"*, continues with the location of the device within the bus system, and ends with an optional *"::INSTR"*.

Possible keyword arguments are:

Keyword	Description
timeout	timeout in seconds for all device operations, see section *Timeouts*. Default: 5
chunk_size	Length of read data chunks in bytes, see section *Chunk length*. Default: 20kB
val-ues_format	Data format for lists of read values, see section *Reading binary data*. Default: *ascii*
term_char	termination characters, see section *Termination characters*. Default: *None*
send_end	whether to assert END after each write operation, see section *Termination characters*. Default: *True*
delay	delay in seconds after each write operation, see section *Termination characters*. Default: 0
lock	whether you want to have exclusive access to the device. Default: *VI_NO_LOCK*

For further information about the locking mechanism, see The VISA library implementation.

The class `Instrument` defines the following methods and attributes:

`Instrument.`**`write`**(*message*)

writes the string *message* to the instrument.

`Instrument.`**`read`**()

returns a string sent from the instrument to the computer.

`Instrument.`**`read_values`**([*format*])

returns a list of decimal values (floats) sent from the instrument to the computer. See section *A more complex example* above. The list may contain only one element or may be empty.

The optional *format* argument overrides the setting of *values_format*. For information about that, see section *Reading binary data*.

`Instrument.`**`ask`**(*message*)

sends the string *message* to the instrument and returns the answer string from the instrument.

`Instrument.`**`ask_for_values`**(*message*[, *format*])

sends the string *message* to the instrument and reads the answer as a list of values, just as *read_values()* does.

The optional *format* argument overrides the setting of *values_format*. For information about that, see section *Reading binary data*.

`Instrument.`**`clear`**()

resets the device. This operation is highly bus-dependent. I refer you to the original VISA documentation, which explains how this is achieved for VXI, GPIB, serial, etc.

`Instrument.`**`trigger`**()

sends a trigger signal to the instrument.

`Instrument.`**`read_raw`**()

returns a string sent from the instrument to the computer. In contrast to *read()*, no termination characters are checked or stripped. You get the pristine message.

`Instrument.timeout`
: The timeout in seconds for each I/O operation. See section *Timeouts* for further information.

`Instrument.term_chars`
: The termination characters for each read and write operation. See section *Termination characters* for further information.

`Instrument.send_end`
: Whether or not to assert EOI (or something equivalent, depending on the interface type) after each write operation. See section *Termination characters* for further information.

`Instrument.delay`
: Time in seconds to wait after each write operation. See section *Termination characters* for further information.

`Instrument.values_format`
: The format for multi-value data sent from the instrument to the computer. See section *Reading binary data* for further information.

2.5.1 Common properties of instrument variables

Timeouts

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.

For all PyVISA objects, a timeout is set with

```
my_device.timeout = 25
```

Here, *my_device* may be a device, an interface or whatever, and its timeout is set to 25 seconds. Floating-point values are allowed. If you set it to zero, all operations must succeed instantaneously. You must not set it to *None*. Instead, if you want to remove the timeout, just say

```
del my_device.timeout
```

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

The default timeout is 5 seconds, but you can change it when creating the device object:

```
my_instrument = instrument("ASRL1", timeout = 8)
```

This creates the object variable *my_instrument* and sets its timeout to 8 seconds. In this context, a timeout value of *None* is allowed, which removes the timeout for this device.

Note that your local VISA library may round up this value heavily. I experienced this effect with my National Instruments VISA implementation, which rounds off to 0, 1, 3 and 10 seconds.

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.

2.5.2 Reading binary data

Some instruments allow for sending the measured data in binary form. This has the advantage that the data transfer is much smaller and takes less time. PyVISA currently supports three forms of transfers:

ascii This is the default mode. It assumes a normal string with comma- or whitespace-separated values.

single The values are expected as a binary sequence of IEEE floating point values with single precision (i.e. four bytes each). All flavours of binary data streams defined in IEEE488.2 are supported, i.e. those beginning with <*header*>#<*digit*>, where <*header*> is optional, and <*digit*> may also be "0".

double The same as **single**, but with values of double precision (eight bytes each).

You can set the form of transfer with the property *values_format*, either with the generation of the object,

```
from pyvisa.highlevel import ascii, single, double

my_instrument = instrument("GPIB::12", values_format = single)
```

or later by setting the property directly:

```
my_instrument.values_format = single
```

Setting this option affects the methods *read_values()* and *ask_for_values()*. In particular, you must assure separately that the device actually sends in this format. In some cases it may be necessary to set the *byte order*, also known as *endianness*. PyVISA assumes little-endian as default. Some instruments call this "swapped" byte order. However, there is also big-endian byte order. In this case you have to append | *big_endian* to your values format:

```
my_instrument = instrument("GPIB::12", values_format = single | big_endian)
```

Example

In order to demonstrate how easy reading binary data can be, remember our example from section *A more complex example*. You just have to append the lines

```
keithley.write("format:data sreal")
keithley.values_format = single
```

to the initialisation commands, and all measurement data will be transmitted as binary. You will only notice the increased speed, as PyVISA converts it into the same list of values as before.

2.5.3 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.term_chars = CR
```

Alternatively you can give it when creating your instrument object:

```
my_instrument = instrument("GPIB::10", term_chars = CR)
```

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

Well, the real default is not *""* (the empty string) but *None*. There is a subtle difference: *""* really means the termination characters are not used at all, neither for read nor for write operations. In contrast, *None* means that every write operation is implicitly terminated with *CR+LF* . This works well with most instruments.

All CRs and LFs are stripped from the end of a read string, no matter how *term_chars* is set.

The termination characters sequence is an ordinary string. *CR* and *LF* are just string constants that allow readable access to *"\r"* and *"\n"*. Therefore, instead of *CR+LF*, you can also write *"\r\n"*, whichever you like more.

delay and *send_end*

There are two further options related to message termination, namely *send_end* and *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 *delay* is the time in seconds to wait after each write operation. So you could write:

```
my_instrument = instrument("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.

2.5.4 GPIB devices

class `GpibInstrument` (*gpib_identifier*[, *board_number*[, ***keyw*]])

represents a GPIB instrument. If *gpib_identifier* is a string, it is interpreted as a VISA resource name (section *VISA resource names*). If it is a number, it denotes the device number at the GPIB bus.

The optional *board_number* defaults to zero. If you have more that one GPIB bus system attached to the computer, you can select the bus with this parameter.

The keyword arguments are interpreted the same as with the class `Instrument`.

Note: Since this class is derived from the class `Instrument`, please refer to section *General devices* for the basic operations. `GpibInstrument` can do everything that `Instrument` can do, so it simply extends the original class with GPIB-specific operations.

The class `GpibInstrument` defines the following methods:

`GpibInstrument.`**`wait_for_srq`**([*timeout*])

waits 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.

The *timeout* argument, given in seconds, denotes the maximal waiting time. The default value is 25 (seconds). If you pass *None* for the timeout, this method waits forever if no SRQ arrives.

class **`Gpib`**(*[board_number]*)

represents a GPIB board. Although most setups have at most one GPIB interface card or USB-GPIB device (with board number 0), theoretically you may have more. Be that as it may, for board-level operations, i.e. operations that affect the whole bus with all connected devices, you must create an instance of this class.

The optional GPIB board number *board_number* defaults to 0.

The class `Gpib` defines the following method:

`Gpib.`**`send_ifc`**()

pulses the interface clear line (IFC) for at least 0.1 seconds.

Note: You needn't store the board instance in a variable. Instead, you may send an IFC signal just by saying *Gpib().send_ifc()*.

2.5.5 Serial devices

Please note that "serial instrument" means only RS232 and parallel port instruments, i.e. everything attached to COM and LPT. In particular, it does not include USB instruments. For USB you have to use `Instrument` instead.

class **`SerialInstrument`**(*resource_name[, **keyw]*)

represents a serial instrument. *resource_name* is the VISA resource name, see section *VISA resource names*. The general keyword arguments are interpreted the same as with the class `Instrument`. The only difference is the default value for *term_chars*: For serial instruments, *CR* (carriage return) is used to terminate readings and writings.

Note: Since this class is derived from the class `Instrument`, please refer to section *General devices* for all operations. `SerialInstrument` can do everything that `Instrument` can do.

The class `SerialInstrument` defines the following additional properties. Note that all properties can also be given as keyword arguments when calling the class constructor or `instrument()`.

`SerialInstrument.`**`baud_rate`**

The communication speed in baud. The default value is 9600.

`SerialInstrument.`**`data_bits`**

Number of data bits contained in each frame. Its value must be from 5 to 8. The default is 8.

`SerialInstrument.`**`stop_bits`**

Number of stop bits contained in each frame. Possible values are 1, 1.5, and 2. The default is 1.

`SerialInstrument.`**`parity`**

The parity used with every frame transmitted and received. Possible values are:

Value	Description
no_parity	no parity bit is used
odd_parity	the parity bit causes odd parity
even_parity	the parity bit causes even parity
mark_parity	the parity bit exists but it's always 1
space_parity	the parity bit exists but it's always 0

The default value is *no_parity*.

`SerialInstrument.`**`end_input`**

This determines the method used to terminate read operations. Possible values are:

Value	Description
last_bit_end_input	read will terminate as soon as a character arrives with its last data bit set
term_chars_end_input	read will terminate as soon as the last character of *term_chars* is received

The default value is *term_chars_end_input*.

2.6 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.

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 such as memory moving.

3. High-level: An object-oriented layer.

   It exposes all functionality using three main clases: *VisaLibrary*, *ResourceManager* and *Instrument*.

It is important to notice that you do not need to import functions from levels 1 and 2, but you can call them directly from the the *VisaLibrary* object. Indeed, all level 1 functions are static methods of *VisaLibrary*. All level 2 functions are bound methods of *VisaLibrary*.

Levels 1 and 2 are implemented in the same package called *ctwrapper* (which stands for ctypes wrapper). The higher level uses *ctwrapper* but in principle can use any package. This will allow us to create other wrappers.

We have two wrappers 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.

CHAPTER 3

More information

3.1 Migrating from PyVISA < 1.5

You don't need to change anything in your code if you only use the *instrument* constructor; and attributes and methods of the resulting object. For example, this code will run unchanged in modern versions of PyVISA:

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

This covers almost every single program that I have seen on the internet. However, if you use other parts of PyVISA or you are interested in the design decisions behind the new version you might want to read on.

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

3.1.1 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.

If you are doing:

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

change it to:

```
>>> import visa
>>> rm = visa.ResourceManager()
>>> keithley = rm.get_instrument("GPIB::12")
>>> print(keithley.ask("*IDN?"))
```

If you are doing:

```
>>> print(visa.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 visa
>>> lib = visa.VisaLibrary("/path/to/my/libvisa.so.7")
```

If you are doing::

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

change it to:

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

If you are doing::

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

change it to:

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

As you see, most of the code shown above is making a few things explict. It adds 1 line of code (instantiating the VisaLibrary or 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.ask_delay* (see below).

3.1.2 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 implementation 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 it will be 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 such as *VisaLibrary*, *VisaException* and error messages. The VISA library was wrapped using ctypes.

In 1.5, we refactored it as *ctwrapper*, also a ctypes wrapper module but it only depends on the constants definitions (*constants.py*). 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.

We have two modules 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 but it will be removed in the future.

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 using the new *VisaLibrary* object 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.

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 = visa.VisaLibrary("/path/to/my/libvisa.so.7")
>>> lib2 = visa.VisaLibrary("/path/to/my/libvisa.so.7")
>>> lib3 = visa.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 = visa.VisaLibrary("/path/to/my/libvisa.so.7")
>>> rm = lib.resource_manager
```

or equivalently:

```
>>> rm = visa.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.

You can still access the legacy classes and global objects:

```
>>> from pyvisa.legacy import vpp43
>>> from pyvisa.legacy import visa_library, resource_manager
```

In 1.5, *visa_library* and *resource_manager*, instances of the legacy classes, will be instantiated on import.

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 visa
>>> lib = visa.VisaLibrary("/path/to/my/libvisa.so.7")
>>> print(lib.read_stb(session))
```

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

```
>>> lib = visa.VisaLibrary("/path/to/my/libvisa.so.7")
>>> status = ctypes.c_ushort()
>>> ret library.viReadSTB(session, ctypes.byref(status))
>>> print(ret.value)
```

Removal of Instrument.delay and added Instrument.ask_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 1.5, *delay* is removed. Delays after write operations must be added to the application code. Instead, a new attribute and argument *ask_delay* is available. This allows you to pause between *write* and *read* operations inside *ask*. Additionally, *ask* takes an optional argument called *delay* 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 1.5, *term_chars* is replaced by *read_termination* and *write_termination*. In this way, you can set independently the termination for each operation. *term_chars* is still present in 1.5 (but will be removed) and sets both at the same time. Automatic removal of *CR + LF* is still present in 1.5 but will be removed in 1.6.

3.2 Contributing to PyVISA

You can contribute in different ways:

3.2.1 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:

```
python -c "from pyvisa import util; util.get_debug_info()"
```

3.2.2 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.

3.3 Frequently asked questions

3.3.1 Is *PyVISA* endorsed by National Instruments?

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

3.3.2 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 is currently maintained by Hernan E. Grecco <hernan.grecco@gmail.com>.

Take a look at AUTHORS for more information

3.3.3 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:

```
python -c "from pyvisa import util; util.get_debug_info()"
```

3.3.4 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

3.3.5 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 by create a configuration file as described in ref:*configuring*.

3.3.6 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/libvis
```

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

```
python -c "from pyvisa import util; util.get_debug_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 ref:*configuring*.

   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.

3.3.7 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

3.4 API

3.4.1 Highlevel module

class `pyvisa.highlevel.`**`VisaLibrary`**

High level VISA Library wrapper.

The easiest way to instantiate the library is to let *pyvisa* find the right one for you. This looks first in your configuration file (~/.pyvisarc). If it fails, it uses *ctypes.util.find_library* to try to locate a library in a way similar to what the compiler does:

```
>>> visa_library = VisaLibrary()
```

But you can also specify the path:

```
>>> visa_library = VisaLibrary('/my/path/visa.so')
```

Or use the *from_paths* constructor if you want to try multiple paths:

```
>>> visa_library = VisaLibrary.from_paths(['/my/path/visa.so', '/maybe/this/visa.so'])
```

Parameters **library_path** – path of the VISA library.

`assert_interrupt_signal`(*library*, *session*, *mode*, *status_id*)

Asserts the specified interrupt or signal.

Parameters

- **library** – the visa library wrapped by ctypes.
- **session** – Unique logical identifier to a session.
- **mode** – How to assert the interrupt. (Constants.ASSERT*)
- **status_id** – This is the status value to be presented during an interrupt acknowledge cycle.

`assert_trigger`(*library*, *session*, *protocol*)

Asserts software or hardware trigger.

Parameters

- **library** – the visa library wrapped by ctypes.
- **session** – Unique logical identifier to a session.
- **protocol** – Trigger protocol to use during assertion. (Constants.PROT*)

`assert_utility_signal`(*library*, *session*, *line*)

Asserts or deasserts the specified utility bus signal.

Parameters

- **library** – the visa library wrapped by ctypes.
- **session** – Unique logical identifier to a session.
- **line** – specifies the utility bus signal to assert. (Constants.UTIL_ASSERT*)

`buffer_read` (*library*, *session*, *count*)
Reads data from device or interface through the use of a formatted I/O read buffer.

Parameters

- **library** – the visa library wrapped by ctypes.
- **session** – Unique logical identifier to a session.
- **count** – Number of bytes to be read.

Returns data read.

Return type bytes

`buffer_write` (*library*, *session*, *data*)
Writes data to a formatted I/O write buffer synchronously.

Parameters

- **library** – the visa library wrapped by ctypes.
- **session** – Unique logical identifier to a session.
- **data** (*bytes*) – data to be written.

Returns number of written bytes.

`clear` (*library*, *session*)
Clears a device.

Parameters

- **library** – the visa library wrapped by ctypes.
- **session** – Unique logical identifier to a session.

`close` (*library*, *session*)
Closes the specified session, event, or find list.

Parameters

- **library** – the visa library wrapped by ctypes.
- **session** – Unique logical identifier to a session, event, or find list.

`disable_event` (*library*, *session*, *event_type*, *mechanism*)
Disables notification of the specified event type(s) via the specified mechanism(s).

Parameters

- **library** – the visa library wrapped by ctypes.
- **session** – Unique logical identifier to a session.
- **event_type** – Logical event identifier.
- **mechanism** – Specifies event handling mechanisms to be disabled. (Constants.QUEUE, .Handler, .SUSPEND_HNDLR, .ALL_MECH)

`discard_events` (*library*, *session*, *event_type*, *mechanism*)
Discards event occurrences for specified event types and mechanisms in a session.

Parameters

- **library** – the visa library wrapped by ctypes.
- **session** – Unique logical identifier to a session.

- **event_type** – Logical event identifier.
- **mechanism** – Specifies event handling mechanisms to be disabled. (Constants.QUEUE, .Handler, .SUSPEND_HNDLR, .ALL_MECH)

`enable_event` (*library*, *session*, *event_type*, *mechanism*, *context=0*)
Enable event occurrences for specified event types and mechanisms in a session.

Parameters

- **library** – the visa library wrapped by ctypes.
- **session** – Unique logical identifier to a session.
- **event_type** – Logical event identifier.
- **mechanism** – Specifies event handling mechanisms to be disabled. (Constants.QUEUE, .Handler, .SUSPEND_HNDLR)
- **context** –

`find_next` (*library*, *find_list*)
Returns the next resource from the list of resources found during a previous call to find_resources().

Parameters

- **library** – the visa library wrapped by ctypes.
- **find_list** – Describes a find list. This parameter must be created by find_resources().

Returns Returns a string identifying the location of a device.

Return type unicode (Py2) or str (Py3)

`find_resources` (*library*, *session*, *query*)
Queries a VISA system to locate the resources associated with a specified interface.

Parameters

- **library** – the visa library wrapped by ctypes.
- **session** – Unique logical identifier to a session (unused, just to uniform signatures).
- **query** – A regular expression followed by an optional logical expression. Use '?*' for all.

Returns find_list, return_counter, instrument_description

Return type ViFindList, int, unicode (Py2) or str (Py3)

`flush` (*library*, *session*, *mask*)
Manually flushes the specified buffers associated with formatted I/O operations and/or serial communication.

Parameters

- **library** – the visa library wrapped by ctypes.
- **session** – Unique logical identifier to a session.
- **mask** – Specifies the action to be taken with flushing the buffer. (Constants.READ*, .WRITE*, .IO*)

classmethod `from_paths` (**paths*)
Helper constructor that tries to instantiate VisaLibrary from an iterable of possible library paths.

`get_attribute` (*library*, *session*, *attribute*)
Retrieves the state of an attribute.

Parameters

- **library** – the visa library wrapped by ctypes.
- **session** – Unique logical identifier to a session, event, or find list.
- **attribute** – Resource attribute for which the state query is made (see Attributes.*)

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

Return type unicode (Py2) or str (Py3), list or other type

`get_default_resource_manager` (*library*)
This function returns a session to the Default Resource Manager resource.

Parameters **library** – the visa library wrapped by ctypes.

Returns Unique logical identifier to a Default Resource Manager session.

`gpib_command` (*library*, *session*, *data*)
Write GPIB command bytes on the bus.

Parameters

- **library** – the visa library wrapped by ctypes.
- **session** – Unique logical identifier to a session.
- **data** (*bytes*) – data tor write.

Returns Number of written bytes.

`gpib_control_atn` (*library*, *session*, *mode*)
Specifies the state of the ATN line and the local active controller state.

Parameters

- **library** – the visa library wrapped by ctypes.
- **session** – Unique logical identifier to a session.
- **mode** – Specifies the state of the ATN line and optionally the local active controller state. (Constants.GPIB_ATN*)

`gpib_control_ren` (*library*, *session*, *mode*)
Controls the state of the GPIB Remote Enable (REN) interface line, and optionally the remote/local state of the device.

Parameters

- **library** – the visa library wrapped by ctypes.
- **session** – Unique logical identifier to a session.
- **mode** – Specifies the state of the REN line and optionally the device remote/local state. (Constants.GPIB_REN*)

`gpib_pass_control` (*library*, *session*, *primary_address*, *secondary_address*)
Tell the GPIB device at the specified address to become controller in charge (CIC).

Parameters

- **library** – the visa library wrapped by ctypes.
- **session** – Unique logical identifier to a session.
- **primary_address** – Primary address of the GPIB device to which you want to pass control.

- **secondary_address** – 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.

`gpib_send_ifc` (*library*, *session*)
Pulse the interface clear line (IFC) for at least 100 microseconds.

Parameters

- **library** – the visa library wrapped by ctypes.
- **session** – Unique logical identifier to a session.

`in_16` (*library*, *session*, *space*, *offset*, *extended=False*)
Reads in an 16-bit value from the specified memory space and offset.

Parameters

- **library** – the visa library wrapped by ctypes.
- **session** – Unique logical identifier to a session.
- **space** – Specifies the address space. (Constants.*SPACE*)
- **offset** – Offset (in bytes) of the address or register from which to read.
- **extended** – Use 64 bits offset independent of the platform.

Returns Data read from memory.

`in_32` (*library*, *session*, *space*, *offset*, *extended=False*)
Reads in an 32-bit value from the specified memory space and offset.

Parameters

- **library** – the visa library wrapped by ctypes.
- **session** – Unique logical identifier to a session.
- **space** – Specifies the address space. (Constants.*SPACE*)
- **offset** – Offset (in bytes) of the address or register from which to read.
- **extended** – Use 64 bits offset independent of the platform.

Returns Data read from memory.

`in_8` (*library*, *session*, *space*, *offset*, *extended=False*)
Reads in an 8-bit value from the specified memory space and offset.

Parameters

- **library** – the visa library wrapped by ctypes.
- **session** – Unique logical identifier to a session.
- **space** – Specifies the address space. (Constants.*SPACE*)
- **offset** – Offset (in bytes) of the address or register from which to read.
- **extended** – Use 64 bits offset independent of the platform.

Returns Data read from memory.

`install_handler` (*session*, *event_type*, *handler*, *user_handle=None*)
Installs handlers for event callbacks.

Parameters

- **session** – Unique logical identifier to a session.
- **event_type** – Logical event identifier.
- **handler** – Interpreted as a valid reference to a 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 user handle (a ctypes object)

`lock` (*library*, *session*, *lock_type*, *timeout*, *requested_key=None*)
Establishes an access mode to the specified resources.

Parameters

- **library** – the visa library wrapped by ctypes.
- **session** – Unique logical identifier to a session.
- **lock_type** – Specifies the type of lock requested, either Constants.EXCLUSIVE_LOCK or Constants.SHARED_LOCK.
- **timeout** – Absolute time period (in milliseconds) that a resource waits to get unlocked by the locking session before returning an error.
- **requested_key** – This parameter is not used and should be set to VI_NULL when lockType is VI_EXCLUSIVE_LOCK.

Returns access_key that can then be passed to other sessions to share the lock.

`map_address` (*library*, *session*, *map_space*, *map_base*, *map_size*, *access=0*, *suggested=0*)
Maps the specified memory space into the process's address space.

Parameters

- **library** – the visa library wrapped by ctypes.
- **session** – Unique logical identifier to a session.
- **map_space** – Specifies the address space to map. (Constants.*SPACE*)
- **map_base** – Offset (in bytes) of the memory to be mapped.
- **map_size** – Amount of memory to map (in bytes).
- **access** –
- **suggested** – If not Constants.NULL (0), the operating system attempts to map the memory to the address specified in suggested. 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 suggested.

Returns Address in your process space where the memory was mapped.

`map_trigger` (*library*, *session*, *trigger_source*, *trigger_destination*, *mode*)
Map the specified trigger source line to the specified destination line.

Parameters

- **library** – the visa library wrapped by ctypes.
- **session** – Unique logical identifier to a session.
- **trigger_source** – Source line from which to map. (Constants.TRIG*)
- **trigger_destination** – Destination line to which to map. (Constants.TRIG*)

- **mode** –

`memory_allocation`(*library*, *session*, *size*, *extended=False*)
Allocates memory from a resource's memory region.

Parameters

- **library** – the visa library wrapped by ctypes.
- **session** – Unique logical identifier to a session.
- **size** – Specifies the size of the allocation.
- **extended** – Use 64 bits offset independent of the platform.

Returns Returns the offset of the allocated memory.

`memory_free`(*library*, *session*, *offset*, *extended=False*)
Frees memory previously allocated using the memory_allocation() operation.

Parameters

- **library** – the visa library wrapped by ctypes.
- **session** – Unique logical identifier to a session.
- **offset** – Offset of the memory to free.
- **extended** – Use 64 bits offset independent of the platform.

`move`(*library*, *session*, *source_space*, *source_offset*, *source_width*, *destination_space*, *destination_offset*, *destination_width*, *length*)
Moves a block of data.

Parameters

- **library** – the visa library wrapped by ctypes.
- **session** – Unique logical identifier to a session.
- **source_space** – Specifies the address space of the source.
- **source_offset** – Offset of the starting address or register from which to read.
- **source_width** – Specifies the data width of the source.
- **destination_space** – Specifies the address space of the destination.
- **destination_offset** – Offset of the starting address or register to which to write.
- **destination_width** – Specifies the data width of the destination.
- **length** – Number of elements to transfer, where the data width of the elements to transfer is identical to the source data width.

`move_asynchronously`(*library*, *session*, *source_space*, *source_offset*, *source_width*, *destination_space*, *destination_offset*, *destination_width*, *length*)
Moves a block of data asynchronously.

Parameters

- **library** – the visa library wrapped by ctypes.
- **session** – Unique logical identifier to a session.
- **source_space** – Specifies the address space of the source.
- **source_offset** – Offset of the starting address or register from which to read.
- **source_width** – Specifies the data width of the source.

- **destination_space** – Specifies the address space of the destination.
- **destination_offset** – Offset of the starting address or register to which to write.
- **destination_width** – Specifies the data width of the destination.
- **length** – Number of elements to transfer, where the data width of the elements to transfer is identical to the source data width.

Returns Job identifier of this asynchronous move operation.

`move_in_16` (*library*, *session*, *space*, *offset*, *length*, *extended=False*)
Moves an 16-bit block of data from the specified address space and offset to local memory.

Parameters

- **library** – the visa library wrapped by ctypes.
- **session** – Unique logical identifier to a session.
- **space** – Specifies the address space. (Constants.*SPACE*)
- **offset** – Offset (in bytes) of the address or register from which to read.
- **length** – Number of elements to transfer, where the data width of the elements to transfer is identical to the source data width.
- **extended** – Use 64 bits offset independent of the platform.

Returns Data read from bus.

Corresponds to viMoveIn16 functions of the visa library.

`move_in_32` (*library*, *session*, *space*, *offset*, *length*, *extended=False*)
Moves an 32-bit block of data from the specified address space and offset to local memory.

Parameters

- **library** – the visa library wrapped by ctypes.
- **session** – Unique logical identifier to a session.
- **space** – Specifies the address space. (Constants.*SPACE*)
- **offset** – Offset (in bytes) of the address or register from which to read.
- **length** – Number of elements to transfer, where the data width of the elements to transfer is identical to the source data width.
- **extended** – Use 64 bits offset independent of the platform.

Returns Data read from bus.

Corresponds to viMoveIn32 functions of the visa library.

`move_in_8` (*library*, *session*, *space*, *offset*, *length*, *extended=False*)
Moves an 8-bit block of data from the specified address space and offset to local memory.

Parameters

- **library** – the visa library wrapped by ctypes.
- **session** – Unique logical identifier to a session.
- **space** – Specifies the address space. (Constants.*SPACE*)
- **offset** – Offset (in bytes) of the address or register from which to read.

- **length** – Number of elements to transfer, where the data width of the elements to transfer is identical to the source data width.
- **extended** – Use 64 bits offset independent of the platform.

Returns Data read from bus.

Corresponds to viMoveIn8 functions of the visa library.

`move_out_16`(*library*, *session*, *space*, *offset*, *length*, *data*, *extended=False*)
Moves an 16-bit block of data from local memory to the specified address space and offset.

Parameters

- **library** – the visa library wrapped by ctypes.
- **session** – Unique logical identifier to a session.
- **space** – Specifies the address space. (Constants.*SPACE*)
- **offset** – Offset (in bytes) of the address or register from which to read.
- **length** – Number of elements to transfer, where the data width of the elements to transfer is identical to the source data width.
- **data** – Data to write to bus.
- **extended** – Use 64 bits offset independent of the platform.

Corresponds to viMoveOut16 functions of the visa library.

`move_out_32`(*library*, *session*, *space*, *offset*, *length*, *data*, *extended=False*)
Moves an 32-bit block of data from local memory to the specified address space and offset.

Parameters

- **library** – the visa library wrapped by ctypes.
- **session** – Unique logical identifier to a session.
- **space** – Specifies the address space. (Constants.*SPACE*)
- **offset** – Offset (in bytes) of the address or register from which to read.
- **length** – Number of elements to transfer, where the data width of the elements to transfer is identical to the source data width.
- **data** – Data to write to bus.
- **extended** – Use 64 bits offset independent of the platform.

Corresponds to viMoveOut32 functions of the visa library.

`move_out_8`(*library*, *session*, *space*, *offset*, *length*, *data*, *extended=False*)
Moves an 8-bit block of data from local memory to the specified address space and offset.

Parameters

- **library** – the visa library wrapped by ctypes.
- **session** – Unique logical identifier to a session.
- **space** – Specifies the address space. (Constants.*SPACE*)
- **offset** – Offset (in bytes) of the address or register from which to read.
- **length** – Number of elements to transfer, where the data width of the elements to transfer is identical to the source data width.

- **data** – Data to write to bus.
- **extended** – Use 64 bits offset independent of the platform.

Corresponds to viMoveOut8 functions of the visa library.

`open` (*library*, *session*, *resource_name*, *access_mode=0*, *open_timeout=0*)

Opens a session to the specified resource.

Parameters

- **library** – the visa library wrapped by ctypes.
- **session** – Resource Manager session (should always be a session returned from open_default_resource_manager()).
- **resource_name** – Unique symbolic name of a resource.
- **access_mode** – Specifies the mode by which the resource is to be accessed. (Constants.NULL or Constants.*LOCK*)
- **open_timeout** – Specifies the maximum time period (in milliseconds) that this operation waits before returning an error.

Returns Unique logical identifier reference to a session.

`open_default_resource_manager` (*library*)

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

Parameters **library** – the visa library wrapped by ctypes.

Returns Unique logical identifier to a Default Resource Manager session.

`out_16` (*library*, *session*, *space*, *offset*, *data*, *extended=False*)

Write in an 16-bit value from the specified memory space and offset. :param library: the visa library wrapped by ctypes. :param session: Unique logical identifier to a session. :param space: Specifies the address space. (Constants.*SPACE*) :param offset: Offset (in bytes) of the address or register from which to read. :param data: Data to write to bus. :param extended: Use 64 bits offset independent of the platform.

Corresponds to viOut16 functions of the visa library.

`out_32` (*library*, *session*, *space*, *offset*, *data*, *extended=False*)

Write in an 32-bit value from the specified memory space and offset. :param library: the visa library wrapped by ctypes. :param session: Unique logical identifier to a session. :param space: Specifies the address space. (Constants.*SPACE*) :param offset: Offset (in bytes) of the address or register from which to read. :param data: Data to write to bus. :param extended: Use 64 bits offset independent of the platform.

Corresponds to viOut32 functions of the visa library.

`out_8` (*library*, *session*, *space*, *offset*, *data*, *extended=False*)

Write in an 8-bit value from the specified memory space and offset. :param library: the visa library wrapped by ctypes. :param session: Unique logical identifier to a session. :param space: Specifies the address space. (Constants.*SPACE*) :param offset: Offset (in bytes) of the address or register from which to read. :param data: Data to write to bus. :param extended: Use 64 bits offset independent of the platform.

Corresponds to viOut8 functions of the visa library.

`parse_resource` (*library*, *session*, *resource_name*)

Parse a resource string to get the interface information.

Parameters

- **library** – the visa library wrapped by ctypes.

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

Returns Resource information with interface type and board number.

Return type :class:ResourceInfo

`parse_resource_extended`(*library*, *session*, *resource_name*)
Parse a resource string to get extended interface information.

Parameters

- **library** – the visa library wrapped by ctypes.
- **session** – Resource Manager session (should always be the Default Resource Manager for VISA returned from open_default_resource_manager()).
- **resource_name** – Unique symbolic name of a resource.

Returns Resource information.

Return type :class:ResourceInfo

`peek_16`(*library*, *session*, *address*)
Read an 16-bit value from the specified address.

Parameters

- **library** – the visa library wrapped by ctypes.
- **session** – Unique logical identifier to a session.
- **address** – Source address to read the value.

Returns Data read from bus.

Return type bytes

`peek_32`(*library*, *session*, *address*)
Read an 32-bit value from the specified address.

Parameters

- **library** – the visa library wrapped by ctypes.
- **session** – Unique logical identifier to a session.
- **address** – Source address to read the value.

Returns Data read from bus.

Return type bytes

`peek_8`(*library*, *session*, *address*)
Read an 8-bit value from the specified address.

Parameters

- **library** – the visa library wrapped by ctypes.
- **session** – Unique logical identifier to a session.
- **address** – Source address to read the value.

Returns Data read from bus.

Return type bytes

poke_16 (*library*, *session*, *address*, *data*)
Write an 16-bit value from the specified address.

Parameters

- **library** – the visa library wrapped by ctypes.
- **session** – Unique logical identifier to a session.
- **address** – Source address to read the value.
- **data** – value to be written to the bus.

Returns Data read from bus.

poke_32 (*library*, *session*, *address*, *data*)
Write an 32-bit value from the specified address.

Parameters

- **library** – the visa library wrapped by ctypes.
- **session** – Unique logical identifier to a session.
- **address** – Source address to read the value.
- **data** – value to be written to the bus.

Returns Data read from bus.

poke_8 (*library*, *session*, *address*, *data*)
Write an 8-bit value from the specified address.

Parameters

- **library** – the visa library wrapped by ctypes.
- **session** – Unique logical identifier to a session.
- **address** – Source address to read the value.
- **data** – value to be written to the bus.

Returns Data read from bus.

read (*library*, *session*, *count*)
Reads data from device or interface synchronously.

Parameters

- **library** – the visa library wrapped by ctypes.
- **session** – Unique logical identifier to a session.
- **count** – Number of bytes to be read.

Returns data read.

Return type bytes

read_asynchronously (*library*, *session*, *count*)
Reads data from device or interface asynchronously.

Parameters

- **library** – the visa library wrapped by ctypes.
- **session** – Unique logical identifier to a session.
- **count** – Number of bytes to be read.

Returns (ctypes buffer with result, jobid)

`read_stb` (*library*, *session*)
Reads a status byte of the service request.

Parameters

- **library** – the visa library wrapped by ctypes.
- **session** – Unique logical identifier to a session.

Returns Service request status byte.

`read_to_file` (*library*, *session*, *filename*, *count*)
Read data synchronously, and store the transferred data in a file.

Parameters

- **library** – the visa library wrapped by ctypes.
- **session** – Unique logical identifier to a session.
- **filename** – Name of file to which data will be written.
- **count** – Number of bytes to be read.

Returns Number of bytes actually transferred.

`resource_manager`
Default resource manager object for this library.

`set_attribute` (*library*, *session*, *attribute*, *attribute_state*)
Sets the state of an attribute.

Parameters

- **library** – the visa library wrapped by ctypes.
- **session** – Unique logical identifier to a session.
- **attribute** – Attribute for which the state is to be modified. (Attributes.*)
- **attribute_state** – The state of the attribute to be set for the specified object.

`set_buffer` (*library*, *session*, *mask*, *size*)
Sets the size for the formatted I/O and/or low-level I/O communication buffer(s).

Parameters

- **library** – the visa library wrapped by ctypes.
- **session** – Unique logical identifier to a session.
- **mask** – Specifies the type of buffer. (Constants.READ_BUF, .WRITE_BUF, .IO_IN_BUF, .IO_OUT_BUF)
- **size** – The size to be set for the specified buffer(s).

`status`
Last return value of the library.

`status_description` (*library*, *session*, *status*)
Returns a user-readable description of the status code passed to the operation.

Parameters

- **library** – the visa library wrapped by ctypes.
- **session** – Unique logical identifier to a session.

- **status** – Status code to interpret.

Returns The user-readable string interpretation of the status code passed to the operation.

Return type unicode (Py2) or str (Py3)

`terminate` (*library*, *session*, *degree*, *job_id*)
Requests a VISA session to terminate normal execution of an operation.

Parameters

- **library** – the visa library wrapped by ctypes.
- **session** – Unique logical identifier to a session.
- **degree** – Constants.NULL
- **job_id** – Specifies an operation identifier.

`uninstall_handler` (*session*, *event_type*, *handler*, *user_handle=None*)
Uninstalls handlers for events.

Parameters

- **session** – Unique logical identifier to a session.
- **event_type** – Logical event identifier.
- **handler** – Interpreted as a valid reference to a 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.

`unlock` (*library*, *session*)
Relinquishes a lock for the specified resource.

Parameters

- **library** – the visa library wrapped by ctypes.
- **session** – Unique logical identifier to a session.

`unmap_address` (*library*, *session*)
Unmaps memory space previously mapped by map_address().

Parameters

- **library** – the visa library wrapped by ctypes.
- **session** – Unique logical identifier to a session.

`unmap_trigger` (*library*, *session*, *trigger_source*, *trigger_destination*)
Undo a previous map from the specified trigger source line to the specified destination line.

Parameters

- **library** – the visa library wrapped by ctypes.
- **session** – Unique logical identifier to a session.
- **trigger_source** – Source line used in previous map. (Constants.TRIG*)
- **trigger_destination** – Destination line used in previous map. (Constants.TRIG*)

`usb_control_in` (*library*, *session*, *request_type_bitmap_field*, *request_id*, *request_value*, *index*, *length=0*)
Performs a USB control pipe transfer from the device.

Parameters

- **library** – the visa library wrapped by ctypes.
- **session** – Unique logical identifier to a session.
- **request_type_bitmap_field** – bmRequestType parameter of the setup stage of a USB control transfer.
- **request_id** – bRequest parameter of the setup stage of a USB control transfer.
- **request_value** – wValue parameter of the setup stage of a USB control transfer.
- **index** – wIndex parameter of the setup stage of a USB control transfer. This is usually the index of the interface or endpoint.
- **length** – 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

`usb_control_out` (*library*, *session*, *request_type_bitmap_field*, *request_id*, *request_value*, *index*, *data=u''*)

Performs a USB control pipe transfer to the device.

Parameters

- **library** – the visa library wrapped by ctypes.
- **session** – Unique logical identifier to a session.
- **request_type_bitmap_field** – bmRequestType parameter of the setup stage of a USB control transfer.
- **request_id** – bRequest parameter of the setup stage of a USB control transfer.
- **request_value** – wValue parameter of the setup stage of a USB control transfer.
- **index** – wIndex parameter of the setup stage of a USB control transfer. This is usually the index of the interface or endpoint.
- **data** – The data buffer that sends the data in the optional data stage of the control transfer.

`vxi_command_query` (*library*, *session*, *mode*, *command*)

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

Parameters

- **library** – the visa library wrapped by ctypes.
- **session** – Unique logical identifier to a session.
- **mode** – Specifies whether to issue a command and/or retrieve a response. (Constants.VXI_CMD*, .VXI_RESP*)
- **command** – The miscellaneous command to send.

Returns The response retrieved from the device.

`wait_on_event` (*library*, *session*, *in_event_type*, *timeout*)

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

Parameters

- **library** – the visa library wrapped by ctypes.

- **session** – Unique logical identifier to a session.
- **in_event_type** – Logical identifier of the event(s) to wait for.
- **timeout** – 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 Logical identifier of the event actually received, A handle specifying the unique occurrence of an event.

`write`(*library*, *session*, *data*)

Writes data to device or interface synchronously.

Parameters

- **library** – the visa library wrapped by ctypes.
- **session** – Unique logical identifier to a session.
- **data** (*str*) – data to be written.

Returns Number of bytes actually transferred.

`write_asynchronously`(*library*, *session*, *data*)

Writes data to device or interface asynchronously.

Parameters

- **library** – the visa library wrapped by ctypes.
- **session** – Unique logical identifier to a session.
- **data** – data to be written.

Returns Job ID of this asynchronous write operation.

`write_from_file`(*library*, *session*, *filename*, *count*)

Take data from a file and write it out synchronously.

Parameters

- **library** – the visa library wrapped by ctypes.
- **session** – Unique logical identifier to a session.
- **filename** – Name of file from which data will be read.
- **count** – Number of bytes to be written.

Returns Number of bytes actually transferred.

class `pyvisa.highlevel.ResourceManager`

VISA Resource Manager

Parameters **visa_library** – VisaLibrary Instance or path of the VISA library (if not given, the default for the platform will be used).

`get_instrument`(*resource_name*, ***kwargs*)

Return an instrument for the resource name.

Parameters

- **resource_name** – name or alias of the resource to open.
- **kwargs** – keyword arguments to be passed to the instrument constructor.

`list_resources`(*query='?*::INSTR'*)

Returns a tuple of all connected devices matching query.

Parameters **query** – regular expression used to match devices.

`list_resources_info`(*query='?*::INSTR'*)
Returns a dictionary mapping resource names to resource extended information of all connected devices matching query.

Parameters **query** – regular expression used to match devices.

Returns Mapping of resource name to ResourceInfo

Return type dict

`open_resource`(*resource_name*, *access_mode=0*, *open_timeout=0*)
Open the specified resources.

Parameters

- **resource_name** – name or alias of the resource to open.
- **access_mode** – access mode.
- **open_timeout** – time out to open.

Returns Unique logical identifier reference to a session.

`resource_info`(*resource_name*)
Get the extended information of a particular resource

Parameters **resource_name** – Unique symbolic name of a resource.

Return type ResourceInfo

class `pyvisa.highlevel.Instrument`(*resource_name*, *resource_manager=None*, ***kwargs*)
Class for all kinds of Instruments.

It can be instantiated, however, if you want to use special features of a certain interface system (GPIB, USB, RS232, etc), you must instantiate one of its child classes.

Parameters

- **resource_name** – the instrument's resource name or an alias, may be taken from the list from *list_resources* method from a ResourceManager.
- **timeout** – the VISA timeout for each low-level operation in milliseconds.
- **term_chars** – the termination characters for this device.
- **chunk_size** – size of data packets in bytes that are read from the device.
- **lock** – whether you want to have exclusive access to the device. Default: VI_NO_LOCK
- **ask_delay** – waiting time in seconds after each write command. Default: 0.0
- **send_end** – whether to assert end line after each write command. Default: True
- **values_format** – floating point data value format. Default: ascii (0)

`ask`(*message*, *delay=None*)
A combination of write(message) and read()

Parameters

- **message** (*str*) – the message to send.
- **delay** – delay in seconds between write and read operations. if None, defaults to self.ask_delay

Returns the answer from the device.

Return type str

`ask_for_values` (*message*, *format=None*, *delay=None*)
A combination of write(message) and read_values()

Parameters

- **message** (*str*) – the message to send.
- **delay** – delay in seconds between write and read operations. if None, defaults to self.ask_delay

Returns the answer from the device.

Return type list

`encoding`
Encoding used for read and write operations.

`read` (*termination=None*, *encoding=None*)
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.

All line-ending characters are stripped from the end of the string.

Return type str

`read_raw` (*size=None*)
Read the unmodified string sent from the instrument to the computer.

In contrast to read(), no termination characters are stripped.

Return type bytes

`read_termination`
Read termination character.

`read_values` (*fmt=None*)
Read a list of floating point values from the device.

Parameters fmt – the format of the values. If given, it overrides the class attribute "values_format". Possible values are bitwise disjunctions of the above constants ascii, single, double, and big_endian. Default is ascii.

Returns the list of read values

Return type list

`send_end`
Whether or not to assert EOI (or something equivalent after each write operation.

`term_chars`
Set or read a new termination character sequence (property).

Normally, you just give the new termination sequence, which is appended to each write operation (unless it's already there), and expected as the ending mark during each read operation. A typical example is CR+LF or just CR. If you assign "" to this property, the termination sequence is deleted.

The default is None, which means that CR + LF is appended to each write operation but not expected after each read operation (but stripped if present).

trigger()
Sends a software trigger to the device.

write(*message*, *termination=None*, *encoding=None*)
Write a string message to the device.

The term_chars are appended to it, unless they are already.

Parameters **message** (*unicode (Py2) or str (Py3)*) – the message to be sent.

Returns number of bytes written.

Return type int

write_raw(*message*)
Write a string message to the device.

The term_chars are appended to it, unless they are already.

Parameters **message** (*bytes*) – the message to be sent.

Returns number of bytes written.

Return type int

write_termination
Writer termination character.

class pyvisa.highlevel.**SerialInstrument**(*resource_name*, *resource_manager=None*, ***keyw*)
Class for serial (RS232 or parallel port) instruments. Not USB!

This class extents the Instrument class with special operations and properties of serial instruments.

Parameters **resource_name** – the instrument's resource name or an alias, may be taken from the list from *list_resources* method from a ResourceManager.

Further keyword arguments are passed to the constructor of class Instrument.

baud_rate
The baud rate of the serial instrument.

data_bits
Number of data bits contained in each frame (from 5 to 8).

end_input
indicates the method used to terminate read operations

parity
The parity used with every frame transmitted and received.

stop_bits
Number of stop bits contained in each frame (1, 1.5, or 2).

class pyvisa.highlevel.**GpibInstrument**(*gpib_identifier*, *board_number=0*, *resource_manager=None*, ***keyw*)
Class for GPIB instruments.

This class extents the Instrument class with special operations and properties of GPIB instruments.

Parameters

- **gpib_identifier** – strings are interpreted as instrument's VISA resource name. Numbers are interpreted as GPIB number.
- **board_number** – the number of the GPIB bus.

Further keyword arguments are passed to the constructor of class Instrument.

`stb`
Service request status register.

`wait_for_srq`(*timeout=25*)
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** – the maximum waiting time in seconds. Defaul: 25 (seconds). None means waiting forever if necessary.

3.4.2 Functions in the ctypes wrapper module

pyvisa.wrapper.functions

Defines VPP 4.3.2 wrapping functions, adding signatures to the library.

This file is part of PyVISA.

copyright 2014 by PyVISA Authors, see AUTHORS for more details.

license MIT, see LICENSE for more details.

`pyvisa.ctwrapper.functions.`**`set_signatures`**(*library*, *errcheck=None*)
Set the signatures of most visa functions in the library.

All instrumentation related functions are specified here. String related functions such as *viPrintf* require a cdecl calling convention even in windows and therefore are require a CDLL object. See *set_cdecl_signatures*.

Parameters

- **library** (*ctypes.WinDLL or ctypes.CDLL*) – the visa library wrapped by ctypes.
- **errcheck** – error checking callable used for visa functions that return ViStatus. It should be take three aregument (result, func, arguments). See errcheck in ctypes.

`pyvisa.ctwrapper.functions.`**`set_cdecl_signatures`**(*clibrary*, *errcheck=None*)
Set the signatures of visa functions requiring a cdecl calling convention.

Parameters

- **clibrary** (*ctypes.CDLL*) – the visa library wrapped by ctypes.
- **errcheck** – error checking callable used for visa functions that return ViStatus. It should be take three aregument (result, func, arguments). See errcheck in ctypes.

`pyvisa.ctwrapper.functions.`**`assert_interrupt_signal`**(*library*, *session*, *mode*, *status_id*)
Asserts the specified interrupt or signal.

Parameters

- **library** – the visa library wrapped by ctypes.
- **session** – Unique logical identifier to a session.
- **mode** – How to assert the interrupt. (Constants.ASSERT*)
- **status_id** – This is the status value to be presented during an interrupt acknowledge cycle.

`pyvisa.ctwrapper.functions.`**`assert_trigger`**(*library*, *session*, *protocol*)
Asserts software or hardware trigger.

Parameters

- **library** – the visa library wrapped by ctypes.

- **session** – Unique logical identifier to a session.
- **protocol** – Trigger protocol to use during assertion. (Constants.PROT*)

`pyvisa.ctwrapper.functions.`**`assert_utility_signal`**(*library*, *session*, *line*)
Asserts or deasserts the specified utility bus signal.

Parameters

- **library** – the visa library wrapped by ctypes.
- **session** – Unique logical identifier to a session.
- **line** – specifies the utility bus signal to assert. (Constants.UTIL_ASSERT*)

`pyvisa.ctwrapper.functions.`**`buffer_read`**(*library*, *session*, *count*)
Reads data from device or interface through the use of a formatted I/O read buffer.

Parameters

- **library** – the visa library wrapped by ctypes.
- **session** – Unique logical identifier to a session.
- **count** – Number of bytes to be read.

Returns data read.

Return type bytes

`pyvisa.ctwrapper.functions.`**`buffer_write`**(*library*, *session*, *data*)
Writes data to a formatted I/O write buffer synchronously.

Parameters

- **library** – the visa library wrapped by ctypes.
- **session** – Unique logical identifier to a session.
- **data** (*bytes*) – data to be written.

Returns number of written bytes.

`pyvisa.ctwrapper.functions.`**`clear`**(*library*, *session*)
Clears a device.

Parameters

- **library** – the visa library wrapped by ctypes.
- **session** – Unique logical identifier to a session.

`pyvisa.ctwrapper.functions.`**`close`**(*library*, *session*)
Closes the specified session, event, or find list.

Parameters

- **library** – the visa library wrapped by ctypes.
- **session** – Unique logical identifier to a session, event, or find list.

`pyvisa.ctwrapper.functions.`**`disable_event`**(*library*, *session*, *event_type*, *mechanism*)
Disables notification of the specified event type(s) via the specified mechanism(s).

Parameters

- **library** – the visa library wrapped by ctypes.
- **session** – Unique logical identifier to a session.

- **event_type** – Logical event identifier.
- **mechanism** – Specifies event handling mechanisms to be disabled. (Constants.QUEUE, .Handler, .SUSPEND_HNDLR, .ALL_MECH)

`pyvisa.ctwrapper.functions.`**`discard_events`**(*library*, *session*, *event_type*, *mechanism*)
Discards event occurrences for specified event types and mechanisms in a session.

Parameters

- **library** – the visa library wrapped by ctypes.
- **session** – Unique logical identifier to a session.
- **event_type** – Logical event identifier.
- **mechanism** – Specifies event handling mechanisms to be disabled. (Constants.QUEUE, .Handler, .SUSPEND_HNDLR, .ALL_MECH)

`pyvisa.ctwrapper.functions.`**`enable_event`**(*library*, *session*, *event_type*, *mechanism*, *context=0*)
Enable event occurrences for specified event types and mechanisms in a session.

Parameters

- **library** – the visa library wrapped by ctypes.
- **session** – Unique logical identifier to a session.
- **event_type** – Logical event identifier.
- **mechanism** – Specifies event handling mechanisms to be disabled. (Constants.QUEUE, .Handler, .SUSPEND_HNDLR)
- **context** –

`pyvisa.ctwrapper.functions.`**`find_next`**(*library*, *find_list*)
Returns the next resource from the list of resources found during a previous call to find_resources().

Parameters

- **library** – the visa library wrapped by ctypes.
- **find_list** – Describes a find list. This parameter must be created by find_resources().

Returns Returns a string identifying the location of a device.

Return type unicode (Py2) or str (Py3)

`pyvisa.ctwrapper.functions.`**`find_resources`**(*library*, *session*, *query*)
Queries a VISA system to locate the resources associated with a specified interface.

Parameters

- **library** – the visa library wrapped by ctypes.
- **session** – Unique logical identifier to a session (unused, just to uniform signatures).
- **query** – A regular expression followed by an optional logical expression. Use '?*' for all.

Returns find_list, return_counter, instrument_description

Return type ViFindList, int, unicode (Py2) or str (Py3)

`pyvisa.ctwrapper.functions.`**`flush`**(*library*, *session*, *mask*)
Manually flushes the specified buffers associated with formatted I/O operations and/or serial communication.

Parameters

- **library** – the visa library wrapped by ctypes.
- **session** – Unique logical identifier to a session.
- **mask** – Specifies the action to be taken with flushing the buffer. (Constants.READ*, .WRITE*, .IO*)

`pyvisa.ctwrapper.functions.`**`get_attribute`**(*library*, *session*, *attribute*)

Retrieves the state of an attribute.

Parameters

- **library** – the visa library wrapped by ctypes.
- **session** – Unique logical identifier to a session, event, or find list.
- **attribute** – Resource attribute for which the state query is made (see Attributes.*)

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

Return type unicode (Py2) or str (Py3), list or other type

`pyvisa.ctwrapper.functions.`**`get_default_resource_manager`**(*library*)

A deprecated alias. See VPP-4.3, rule 4.3.5 and observation 4.3.2.

`pyvisa.ctwrapper.functions.`**`gpib_command`**(*library*, *session*, *data*)

Write GPIB command bytes on the bus.

Parameters

- **library** – the visa library wrapped by ctypes.
- **session** – Unique logical identifier to a session.
- **data** (*bytes*) – data tor write.

Returns Number of written bytes.

`pyvisa.ctwrapper.functions.`**`gpib_control_atn`**(*library*, *session*, *mode*)

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

Parameters

- **library** – the visa library wrapped by ctypes.
- **session** – Unique logical identifier to a session.
- **mode** – Specifies the state of the ATN line and optionally the local active controller state. (Constants.GPIB_ATN*)

`pyvisa.ctwrapper.functions.`**`gpib_control_ren`**(*library*, *session*, *mode*)

Controls the state of the GPIB Remote Enable (REN) interface line, and optionally the remote/local state of the device.

Parameters

- **library** – the visa library wrapped by ctypes.
- **session** – Unique logical identifier to a session.
- **mode** – Specifies the state of the REN line and optionally the device remote/local state. (Constants.GPIB_REN*)

`pyvisa.ctwrapper.functions.`**`gpib_pass_control`**(*library*, *session*, *primary_address*, *secondary_address*)

Tell the GPIB device at the specified address to become controller in charge (CIC).

Parameters

- **library** – the visa library wrapped by ctypes.
- **session** – Unique logical identifier to a session.
- **primary_address** – Primary address of the GPIB device to which you want to pass control.
- **secondary_address** – 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.

`pyvisa.ctwrapper.functions.`**`gpib_send_ifc`**(*library*, *session*)

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

Parameters

- **library** – the visa library wrapped by ctypes.
- **session** – Unique logical identifier to a session.

`pyvisa.ctwrapper.functions.`**`in_16`**(*library*, *session*, *space*, *offset*, *extended=False*)

Reads in an 16-bit value from the specified memory space and offset.

Parameters

- **library** – the visa library wrapped by ctypes.
- **session** – Unique logical identifier to a session.
- **space** – Specifies the address space. (Constants.*SPACE*)
- **offset** – Offset (in bytes) of the address or register from which to read.
- **extended** – Use 64 bits offset independent of the platform.

Returns Data read from memory.

`pyvisa.ctwrapper.functions.`**`in_32`**(*library*, *session*, *space*, *offset*, *extended=False*)

Reads in an 32-bit value from the specified memory space and offset.

Parameters

- **library** – the visa library wrapped by ctypes.
- **session** – Unique logical identifier to a session.
- **space** – Specifies the address space. (Constants.*SPACE*)
- **offset** – Offset (in bytes) of the address or register from which to read.
- **extended** – Use 64 bits offset independent of the platform.

Returns Data read from memory.

`pyvisa.ctwrapper.functions.`**`in_8`**(*library*, *session*, *space*, *offset*, *extended=False*)

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

Parameters

- **library** – the visa library wrapped by ctypes.
- **session** – Unique logical identifier to a session.
- **space** – Specifies the address space. (Constants.*SPACE*)
- **offset** – Offset (in bytes) of the address or register from which to read.
- **extended** – Use 64 bits offset independent of the platform.

Returns Data read from memory.

`pyvisa.ctwrapper.functions.`**`install_handler`**(*library*, *session*, *event_type*, *handler*, *user_handle*)

Installs handlers for event callbacks.

Parameters

- **library** – the visa library wrapped by ctypes.
- **session** – Unique logical identifier to a session.
- **event_type** – Logical event identifier.
- **handler** – Interpreted as a valid reference to a 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 a handler descriptor which consists of three elements: - handler (a python callable) - user handle (a ctypes object) - ctypes handler (ctypes object wrapping handler)

`pyvisa.ctwrapper.functions.`**`lock`**(*library*, *session*, *lock_type*, *timeout*, *requested_key=None*)

Establishes an access mode to the specified resources.

Parameters

- **library** – the visa library wrapped by ctypes.
- **session** – Unique logical identifier to a session.
- **lock_type** – Specifies the type of lock requested, either Constants.EXCLUSIVE_LOCK or Constants.SHARED_LOCK.
- **timeout** – Absolute time period (in milliseconds) that a resource waits to get unlocked by the locking session before returning an error.
- **requested_key** – This parameter is not used and should be set to VI_NULL when lockType is VI_EXCLUSIVE_LOCK.

Returns access_key that can then be passed to other sessions to share the lock.

`pyvisa.ctwrapper.functions.`**`map_address`**(*library*, *session*, *map_space*, *map_base*, *map_size*, *access=0*, *suggested=0*)

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

Parameters

- **library** – the visa library wrapped by ctypes.
- **session** – Unique logical identifier to a session.
- **map_space** – Specifies the address space to map. (Constants.*SPACE*)
- **map_base** – Offset (in bytes) of the memory to be mapped.
- **map_size** – Amount of memory to map (in bytes).
- **access** –
- **suggested** – If not Constants.NULL (0), the operating system attempts to map the memory to the address specified in suggested. 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 suggested.

Returns Address in your process space where the memory was mapped.

`pyvisa.ctwrapper.functions.`**`map_trigger`**(*library*, *session*, *trigger_source*, *trigger_destination*, *mode*)

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

Parameters

- **library** – the visa library wrapped by ctypes.
- **session** – Unique logical identifier to a session.
- **trigger_source** – Source line from which to map. (Constants.TRIG*)
- **trigger_destination** – Destination line to which to map. (Constants.TRIG*)
- **mode** –

`pyvisa.ctwrapper.functions.`**`memory_allocation`**(*library*, *session*, *size*, *extended=False*)

Allocates memory from a resource's memory region.

Parameters

- **library** – the visa library wrapped by ctypes.
- **session** – Unique logical identifier to a session.
- **size** – Specifies the size of the allocation.
- **extended** – Use 64 bits offset independent of the platform.

Returns Returns the offset of the allocated memory.

`pyvisa.ctwrapper.functions.`**`memory_free`**(*library*, *session*, *offset*, *extended=False*)

Frees memory previously allocated using the memory_allocation() operation.

Parameters

- **library** – the visa library wrapped by ctypes.
- **session** – Unique logical identifier to a session.
- **offset** – Offset of the memory to free.
- **extended** – Use 64 bits offset independent of the platform.

`pyvisa.ctwrapper.functions.`**`move`**(*library*, *session*, *source_space*, *source_offset*, *source_width*, *destination_space*, *destination_offset*, *destination_width*, *length*)

Moves a block of data.

Parameters

- **library** – the visa library wrapped by ctypes.
- **session** – Unique logical identifier to a session.
- **source_space** – Specifies the address space of the source.
- **source_offset** – Offset of the starting address or register from which to read.
- **source_width** – Specifies the data width of the source.
- **destination_space** – Specifies the address space of the destination.
- **destination_offset** – Offset of the starting address or register to which to write.
- **destination_width** – Specifies the data width of the destination.
- **length** – Number of elements to transfer, where the data width of the elements to transfer is identical to the source data width.

`pyvisa.ctwrapper.functions.`**`move_asynchronously`**(*library*, *session*, *source_space*, *source_offset*, *source_width*, *destination_space*, *destination_offset*, *destination_width*, *length*)

Moves a block of data asynchronously.

Parameters

- **library** – the visa library wrapped by ctypes.
- **session** – Unique logical identifier to a session.
- **source_space** – Specifies the address space of the source.
- **source_offset** – Offset of the starting address or register from which to read.
- **source_width** – Specifies the data width of the source.
- **destination_space** – Specifies the address space of the destination.
- **destination_offset** – Offset of the starting address or register to which to write.
- **destination_width** – Specifies the data width of the destination.
- **length** – Number of elements to transfer, where the data width of the elements to transfer is identical to the source data width.

Returns Job identifier of this asynchronous move operation.

`pyvisa.ctwrapper.functions.`**`move_in_16`**(*library*, *session*, *space*, *offset*, *length*, *extended=False*)

Moves an 16-bit block of data from the specified address space and offset to local memory.

Parameters

- **library** – the visa library wrapped by ctypes.
- **session** – Unique logical identifier to a session.
- **space** – Specifies the address space. (Constants.*SPACE*)
- **offset** – Offset (in bytes) of the address or register from which to read.
- **length** – Number of elements to transfer, where the data width of the elements to transfer is identical to the source data width.
- **extended** – Use 64 bits offset independent of the platform.

Returns Data read from bus.

Corresponds to viMoveIn16 functions of the visa library.

`pyvisa.ctwrapper.functions.`**`move_in_32`**(*library*, *session*, *space*, *offset*, *length*, *extended=False*)

Moves an 32-bit block of data from the specified address space and offset to local memory.

Parameters

- **library** – the visa library wrapped by ctypes.
- **session** – Unique logical identifier to a session.
- **space** – Specifies the address space. (Constants.*SPACE*)
- **offset** – Offset (in bytes) of the address or register from which to read.
- **length** – Number of elements to transfer, where the data width of the elements to transfer is identical to the source data width.

- **extended** – Use 64 bits offset independent of the platform.

Returns Data read from bus.

Corresponds to viMoveIn32 functions of the visa library.

`pyvisa.ctwrapper.functions.`**`move_in_8`**(*library*, *session*, *space*, *offset*, *length*, *extended=False*)

Moves an 8-bit block of data from the specified address space and offset to local memory.

Parameters

- **library** – the visa library wrapped by ctypes.
- **session** – Unique logical identifier to a session.
- **space** – Specifies the address space. (Constants.*SPACE*)
- **offset** – Offset (in bytes) of the address or register from which to read.
- **length** – Number of elements to transfer, where the data width of the elements to transfer is identical to the source data width.
- **extended** – Use 64 bits offset independent of the platform.

Returns Data read from bus.

Corresponds to viMoveIn8 functions of the visa library.

`pyvisa.ctwrapper.functions.`**`move_out_16`**(*library*, *session*, *space*, *offset*, *length*, *data*, *extended=False*)

Moves an 16-bit block of data from local memory to the specified address space and offset.

Parameters

- **library** – the visa library wrapped by ctypes.
- **session** – Unique logical identifier to a session.
- **space** – Specifies the address space. (Constants.*SPACE*)
- **offset** – Offset (in bytes) of the address or register from which to read.
- **length** – Number of elements to transfer, where the data width of the elements to transfer is identical to the source data width.
- **data** – Data to write to bus.
- **extended** – Use 64 bits offset independent of the platform.

Corresponds to viMoveOut16 functions of the visa library.

`pyvisa.ctwrapper.functions.`**`move_out_32`**(*library*, *session*, *space*, *offset*, *length*, *data*, *extended=False*)

Moves an 32-bit block of data from local memory to the specified address space and offset.

Parameters

- **library** – the visa library wrapped by ctypes.
- **session** – Unique logical identifier to a session.
- **space** – Specifies the address space. (Constants.*SPACE*)
- **offset** – Offset (in bytes) of the address or register from which to read.
- **length** – Number of elements to transfer, where the data width of the elements to transfer is identical to the source data width.
- **data** – Data to write to bus.

- **extended** – Use 64 bits offset independent of the platform.

Corresponds to viMoveOut32 functions of the visa library.

`pyvisa.ctwrapper.functions.`**`move_out_8`**(*library*, *session*, *space*, *offset*, *length*, *data*, *extended=False*)

Moves an 8-bit block of data from local memory to the specified address space and offset.

Parameters

- **library** – the visa library wrapped by ctypes.
- **session** – Unique logical identifier to a session.
- **space** – Specifies the address space. (Constants.*SPACE*)
- **offset** – Offset (in bytes) of the address or register from which to read.
- **length** – Number of elements to transfer, where the data width of the elements to transfer is identical to the source data width.
- **data** – Data to write to bus.
- **extended** – Use 64 bits offset independent of the platform.

Corresponds to viMoveOut8 functions of the visa library.

`pyvisa.ctwrapper.functions.`**`open`**(*library*, *session*, *resource_name*, *access_mode=0*, *open_timeout=0*)

Opens a session to the specified resource.

Parameters

- **library** – the visa library wrapped by ctypes.
- **session** – Resource Manager session (should always be a session returned from open_default_resource_manager()).
- **resource_name** – Unique symbolic name of a resource.
- **access_mode** – Specifies the mode by which the resource is to be accessed. (Constants.NULL or Constants.*LOCK*)
- **open_timeout** – Specifies the maximum time period (in milliseconds) that this operation waits before returning an error.

Returns Unique logical identifier reference to a session.

`pyvisa.ctwrapper.functions.`**`open_default_resource_manager`**(*library*)

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

Parameters **library** – the visa library wrapped by ctypes.

Returns Unique logical identifier to a Default Resource Manager session.

`pyvisa.ctwrapper.functions.`**`out_16`**(*library*, *session*, *space*, *offset*, *data*, *extended=False*)

Write in an 16-bit value from the specified memory space and offset. :param library: the visa library wrapped by ctypes. :param session: Unique logical identifier to a session. :param space: Specifies the address space. (Constants.*SPACE*) :param offset: Offset (in bytes) of the address or register from which to read. :param data: Data to write to bus. :param extended: Use 64 bits offset independent of the platform.

Corresponds to viOut16 functions of the visa library.

`pyvisa.ctwrapper.functions.`**`out_32`**(*library*, *session*, *space*, *offset*, *data*, *extended=False*)

Write in an 32-bit value from the specified memory space and offset. :param library: the visa library wrapped by ctypes. :param session: Unique logical identifier to a session. :param space: Specifies the address space.

(Constants.*SPACE*) :param offset: Offset (in bytes) of the address or register from which to read. :param data: Data to write to bus. :param extended: Use 64 bits offset independent of the platform.

Corresponds to viOut32 functions of the visa library.

`pyvisa.ctwrapper.functions.`**`out_8`**(*library*, *session*, *space*, *offset*, *data*, *extended=False*)

Write in an 8-bit value from the specified memory space and offset. :param library: the visa library wrapped by ctypes. :param session: Unique logical identifier to a session. :param space: Specifies the address space. (Constants.*SPACE*) :param offset: Offset (in bytes) of the address or register from which to read. :param data: Data to write to bus. :param extended: Use 64 bits offset independent of the platform.

Corresponds to viOut8 functions of the visa library.

`pyvisa.ctwrapper.functions.`**`parse_resource`**(*library*, *session*, *resource_name*)

Parse a resource string to get the interface information.

Parameters

- **library** – the visa library wrapped by ctypes.
- **session** – Resource Manager session (should always be the Default Resource Manager for VISA returned from open_default_resource_manager()).
- **resource_name** – Unique symbolic name of a resource.

Returns Resource information with interface type and board number.

Return type :class:ResourceInfo

`pyvisa.ctwrapper.functions.`**`parse_resource_extended`**(*library*, *session*, *resource_name*)

Parse a resource string to get extended interface information.

Parameters

- **library** – the visa library wrapped by ctypes.
- **session** – Resource Manager session (should always be the Default Resource Manager for VISA returned from open_default_resource_manager()).
- **resource_name** – Unique symbolic name of a resource.

Returns Resource information.

Return type :class:ResourceInfo

`pyvisa.ctwrapper.functions.`**`peek_16`**(*library*, *session*, *address*)

Read an 16-bit value from the specified address.

Parameters

- **library** – the visa library wrapped by ctypes.
- **session** – Unique logical identifier to a session.
- **address** – Source address to read the value.

Returns Data read from bus.

Return type bytes

`pyvisa.ctwrapper.functions.`**`peek_32`**(*library*, *session*, *address*)

Read an 32-bit value from the specified address.

Parameters

- **library** – the visa library wrapped by ctypes.
- **session** – Unique logical identifier to a session.

- **address** – Source address to read the value.

Returns Data read from bus.

Return type bytes

`pyvisa.ctwrapper.functions.`**`peek_8`**(*library*, *session*, *address*)
Read an 8-bit value from the specified address.

Parameters

- **library** – the visa library wrapped by ctypes.
- **session** – Unique logical identifier to a session.
- **address** – Source address to read the value.

Returns Data read from bus.

Return type bytes

`pyvisa.ctwrapper.functions.`**`poke_16`**(*library*, *session*, *address*, *data*)
Write an 16-bit value from the specified address.

Parameters

- **library** – the visa library wrapped by ctypes.
- **session** – Unique logical identifier to a session.
- **address** – Source address to read the value.
- **data** – value to be written to the bus.

Returns Data read from bus.

`pyvisa.ctwrapper.functions.`**`poke_32`**(*library*, *session*, *address*, *data*)
Write an 32-bit value from the specified address.

Parameters

- **library** – the visa library wrapped by ctypes.
- **session** – Unique logical identifier to a session.
- **address** – Source address to read the value.
- **data** – value to be written to the bus.

Returns Data read from bus.

`pyvisa.ctwrapper.functions.`**`poke_8`**(*library*, *session*, *address*, *data*)
Write an 8-bit value from the specified address.

Parameters

- **library** – the visa library wrapped by ctypes.
- **session** – Unique logical identifier to a session.
- **address** – Source address to read the value.
- **data** – value to be written to the bus.

Returns Data read from bus.

`pyvisa.ctwrapper.functions.`**`read`**(*library*, *session*, *count*)
Reads data from device or interface synchronously.

Parameters

- **library** – the visa library wrapped by ctypes.
- **session** – Unique logical identifier to a session.
- **count** – Number of bytes to be read.

Returns data read.

Return type bytes

`pyvisa.ctwrapper.functions.`**`read_asynchronously`**(*library*, *session*, *count*)
Reads data from device or interface asynchronously.

Parameters

- **library** – the visa library wrapped by ctypes.
- **session** – Unique logical identifier to a session.
- **count** – Number of bytes to be read.

Returns (ctypes buffer with result, jobid)

`pyvisa.ctwrapper.functions.`**`read_to_file`**(*library*, *session*, *filename*, *count*)
Read data synchronously, and store the transferred data in a file.

Parameters

- **library** – the visa library wrapped by ctypes.
- **session** – Unique logical identifier to a session.
- **filename** – Name of file to which data will be written.
- **count** – Number of bytes to be read.

Returns Number of bytes actually transferred.

`pyvisa.ctwrapper.functions.`**`read_stb`**(*library*, *session*)
Reads a status byte of the service request.

Parameters

- **library** – the visa library wrapped by ctypes.
- **session** – Unique logical identifier to a session.

Returns Service request status byte.

`pyvisa.ctwrapper.functions.`**`set_attribute`**(*library*, *session*, *attribute*, *attribute_state*)
Sets the state of an attribute.

Parameters

- **library** – the visa library wrapped by ctypes.
- **session** – Unique logical identifier to a session.
- **attribute** – Attribute for which the state is to be modified. (Attributes.*)
- **attribute_state** – The state of the attribute to be set for the specified object.

`pyvisa.ctwrapper.functions.`**`set_buffer`**(*library*, *session*, *mask*, *size*)
Sets the size for the formatted I/O and/or low-level I/O communication buffer(s).

Parameters

- **library** – the visa library wrapped by ctypes.
- **session** – Unique logical identifier to a session.

- **mask** – Specifies the type of buffer. (Constants.READ_BUF, .WRITE_BUF, .IO_IN_BUF, .IO_OUT_BUF)
- **size** – The size to be set for the specified buffer(s).

`pyvisa.ctwrapper.functions.`**`status_description`**(*library*, *session*, *status*)

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

Parameters

- **library** – the visa library wrapped by ctypes.
- **session** – Unique logical identifier to a session.
- **status** – Status code to interpret.

Returns The user-readable string interpretation of the status code passed to the operation.

Return type unicode (Py2) or str (Py3)

`pyvisa.ctwrapper.functions.`**`terminate`**(*library*, *session*, *degree*, *job_id*)

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

Parameters

- **library** – the visa library wrapped by ctypes.
- **session** – Unique logical identifier to a session.
- **degree** – Constants.NULL
- **job_id** – Specifies an operation identifier.

`pyvisa.ctwrapper.functions.`**`uninstall_handler`**(*library*, *session*, *event_type*, *handler*, *user_handle=None*)

Uninstalls handlers for events.

Parameters

- **library** – the visa library wrapped by ctypes.
- **session** – Unique logical identifier to a session.
- **event_type** – Logical event identifier.
- **handler** – Interpreted as a valid reference to a 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.

`pyvisa.ctwrapper.functions.`**`unlock`**(*library*, *session*)

Relinquishes a lock for the specified resource.

Parameters

- **library** – the visa library wrapped by ctypes.
- **session** – Unique logical identifier to a session.

`pyvisa.ctwrapper.functions.`**`unmap_address`**(*library*, *session*)

Unmaps memory space previously mapped by map_address().

Parameters

- **library** – the visa library wrapped by ctypes.
- **session** – Unique logical identifier to a session.

`pyvisa.ctwrapper.functions.`**`unmap_trigger`**(*library*, *session*, *trigger_source*, *trigger_destination*)

Undo a previous map from the specified trigger source line to the specified destination line.

Parameters

- **library** – the visa library wrapped by ctypes.
- **session** – Unique logical identifier to a session.
- **trigger_source** – Source line used in previous map. (Constants.TRIG*)
- **trigger_destination** – Destination line used in previous map. (Constants.TRIG*)

`pyvisa.ctwrapper.functions.`**`usb_control_in`**(*library*, *session*, *request_type_bitmap_field*, *request_id*, *request_value*, *index*, *length=0*)

Performs a USB control pipe transfer from the device.

Parameters

- **library** – the visa library wrapped by ctypes.
- **session** – Unique logical identifier to a session.
- **request_type_bitmap_field** – bmRequestType parameter of the setup stage of a USB control transfer.
- **request_id** – bRequest parameter of the setup stage of a USB control transfer.
- **request_value** – wValue parameter of the setup stage of a USB control transfer.
- **index** – wIndex parameter of the setup stage of a USB control transfer. This is usually the index of the interface or endpoint.
- **length** – 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

`pyvisa.ctwrapper.functions.`**`usb_control_out`**(*library*, *session*, *request_type_bitmap_field*, *request_id*, *request_value*, *index*, *data=u''*)

Performs a USB control pipe transfer to the device.

Parameters

- **library** – the visa library wrapped by ctypes.
- **session** – Unique logical identifier to a session.
- **request_type_bitmap_field** – bmRequestType parameter of the setup stage of a USB control transfer.
- **request_id** – bRequest parameter of the setup stage of a USB control transfer.
- **request_value** – wValue parameter of the setup stage of a USB control transfer.
- **index** – wIndex parameter of the setup stage of a USB control transfer. This is usually the index of the interface or endpoint.
- **data** – The data buffer that sends the data in the optional data stage of the control transfer.

`pyvisa.ctwrapper.functions.`**`vxi_command_query`**(*library*, *session*, *mode*, *command*)

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

Parameters

- **library** – the visa library wrapped by ctypes.
- **session** – Unique logical identifier to a session.
- **mode** – Specifies whether to issue a command and/or retrieve a response. (Constants.VXI_CMD*, .VXI_RESP*)
- **command** – The miscellaneous command to send.

Returns The response retrieved from the device.

`pyvisa.ctwrapper.functions.`**`wait_on_event`**(*library*, *session*, *in_event_type*, *timeout*)
Waits for an occurrence of the specified event for a given session.

Parameters

- **library** – the visa library wrapped by ctypes.
- **session** – Unique logical identifier to a session.
- **in_event_type** – Logical identifier of the event(s) to wait for.
- **timeout** – 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 Logical identifier of the event actually received, A handle specifying the unique occurrence of an event.

`pyvisa.ctwrapper.functions.`**`write`**(*library*, *session*, *data*)
Writes data to device or interface synchronously.

Parameters

- **library** – the visa library wrapped by ctypes.
- **session** – Unique logical identifier to a session.
- **data** (*str*) – data to be written.

Returns Number of bytes actually transferred.

`pyvisa.ctwrapper.functions.`**`write_asynchronously`**(*library*, *session*, *data*)
Writes data to device or interface asynchronously.

Parameters

- **library** – the visa library wrapped by ctypes.
- **session** – Unique logical identifier to a session.
- **data** – data to be written.

Returns Job ID of this asynchronous write operation.

`pyvisa.ctwrapper.functions.`**`write_from_file`**(*library*, *session*, *filename*, *count*)
Take data from a file and write it out synchronously.

Parameters

- **library** – the visa library wrapped by ctypes.
- **session** – Unique logical identifier to a session.
- **filename** – Name of file from which data will be read.
- **count** – Number of bytes to be written.

Returns Number of bytes actually transferred.

CHAPTER 4

Legacy Modules

Note: This is a legacy module kept for backwards compatiblity with PyVISA < 1.5 and will be deprecated in future versions of PyVISA. You are strongly encouraged to switch to the new implementation.

4.1 About the legacy visa module

Abstract

PyVISA enables you to control your measurement and test equipment – digital multimeters, motors, sensors and the like. This document covers the easy-to- use `visa` module of the PyVISA package. It implements control of measurement devices in a straightforward and convenient way. The design goal is to combine HTBasic's simplicity with Python's modern syntax and powerful set of libraries. PyVISA doesn't implement VISA itself. Instead, PyVISA provides bindings to the VISA library (a DLL or "shared object" file). This library is usually shipped with your GPIB interface or software like LabVIEW . Alternatively, you can download it from your favourite equipment vendor (National Instruments, Agilent, etc).

It can be downloaded at the PyVISA project page. You can report bugs there, too. Additionally, I'm happy about feedback from people who've given it a try. So far, we have positive reports of various National Instruments GPIB adapters (connected through PCI, USB, and RS232), the Agilent 82357A, and SRS lock-in amplifiers, for both Windows and Linux. However, I'd be really surprised about negative reports anyway, due to the high abstraction level of PyVISA . As far as USB instruments are concerned, you must make sure that they act as ordinary USB devices and not as so-called HDI devices (like keyboard and mouse).

Contents

- About the legacy visa module
  - An example
  - Example for serial (RS232) device
  - A more complex example
  - VISA resource names

4.1.1 An example

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

```
from pyvisa.legacy import visa

my_instrument = instrument("GPIB::14")
my_instrument.write("*IDN?")
print my_instrument.read()
```

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

Every instrument is represented in the source by an object instance. In this case, I have a GPIB instrument with instrument number 14, so I create the instance (i.e. variable) called *my_instrument* accordingly:

```
my_instrument = instrument("GPIB::14")
```

"GPIB::14" is the instrument's *resource name*. See section *VISA resource names* for a short explanation of that. Then, I send the message *"*IDN?"* to the device, which is the standard GPIB message for "what are you?" or – in some cases – "what's on your display at the moment?":

```
my_instrument.write("*IDN?")
```

Finally, I print the instrument's answer on the screen:

```
print(my_instrument.read())
```

4.1.2 Example for serial (RS232) device

The only RS232 device in my lab is an old Oxford ITC4 temperature controller, which is connected through COM2 with my computer. The following code prints its self-identification on the screen:

```
from pyvisa.legacy import visa

itc4 = visa.instrument("COM2")
itc4.write("V")
print(itc4.read())
```

Instead of separate write and read operations, you can do both with one *ask()* call. Thus, the above source code is equivalent to:

```
from pyvisa.legacy import visa

itc4 = visa.instrument("COM2")
print(itc4.ask("V"))
```

It couldn't be simpler. See section *Serial devices* for further information about serial devices.

4.1.3 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 initialise the instrument:

```
from pyvisa.legacy import visa

keithley = visa.instrument("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 initialisation 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; 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.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.ask("trace:data?")* however, then we'd get

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

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

```
voltages = keithley.ask_for_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.ask("status:measurement?")
keithley.write("trace:clear; feed:control next")
```

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

4.1.4 VISA resource names

If you use the function `instrument()`, 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.

Note: This is a legacy module kept for backwards compatiblity with PyVISA < 1.5. and will be deprecated in future versions of PyVISA. You are strongly encouraged to switch to the new implementation.

4.2 About the legacy vpp43 module

This module `vpp43` is a cautious yet thorough adaption of the VISA specification for Python. The "textual languages" VISA specification can't be implemented as is because Python is rather different from C and Visual Basic, most notably because of lacking call-by-reference. The second important difference are strings: In C they are null-terminated whereas Python doesn't have this constraint.

The slightly odd name `vpp43` for this module derives from the necessity to make (name)space for the `visa` module that is supposed to realise the actual high-level VISA access in Python. The VXIplug&play Systems Alliance used to maintain the VISA specifications, and, although today the IVI foundation is responsible for this task, the files are still called `vpp43.doc` etc. So I thought `vpp43` was an appropriate name.

You may wonder why I did choose new names for all routines. I did so because Python has its own naming guidelines, and because it shows that the routines had to be adapted. However, I didn't change them really: Every routine is a 1:1 counterpart. By calling them from C, you could even create a C-based VISA implementation with the original function signatures and semantics. Moreover, the new names are mere expansions of the original ones.

4.2.1 Connecting to the VISA shared object

`vpp43` tries to find the VISA library for itself. On Windows, this is not a big problem. `visa32.dll` must be in your `PATH`. If it isn't, move it there or expand your `PATH`.

However, on Linux you may need to give the explicit path to the shared object file. You do so by saying for example:

```python
from pyvisa.legacy import vpp43
vpp43.visa_library.load_library("/path/to/my/libvisa.so.7")
```

By default, `vpp43` looks for the library in `/usr/local/vxipnp/linux/bin/libvisa.so.7`. Please pay attention to the fact that the library must have been successfully loaded *before* any VISA call is made.

Alternatively, you can tell PyVISA so by creating a file `~/.pyvisarc`. This 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.

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.)

4.2.2 Diagnostics

This module can raise a couple of `vpp43`-specific exceptions.

Name VisaIOError

Description This is an error of the underlying VISA library, as described in table 3.3.1 in the VISA specification for textual languages. The exception member `error_code` contains the (always negative) VISA error number, as listed in that table.

Name VisaIOWarning

Description This is a warning of the underlying VISA library, as described in table 3.3.1 in the VISA specification for textual languages. The exception member `completion_code` contains the (always positive) VISA completion number, as listed in that table.

Normally you don't see these warnings. You can turn them into exceptions with:

```
import warnings
warnings.filterwarnings("error")
```

Consult the description of the `warnings` package for further information.

Name TypeError

Description The current implementation of **'printf'_**, **'scanf'_**, **'sprintf'_**, **'sscanf'_**, and **'queryf'_** have the limitation that only integers, floats, and strings are allowed as types for the arbitrary arguments. Additionally, only format string directives for C longs, C doubles, and C strings are allowed to use, albeit not checked. However, if you pass a list or a unicode string, you get this exception.

The same exception is raised if an unsupported type is passed as user handle to **'install_handler'_**. See there for a list of supported types.

Name UnknownHandler

Description Raised if an unknown *handler/user_handle* pair is passed to **'uninstall_handler'_**. In particular, you must save the user handle returned by **'install_handler'_** in order to pass it to uninstall_handler.

Moreover, this module may pass exceptions generated by ctypes. This may be because you've passed a wrong type to a function, or that the VISA library file was not found, but it may also mean a bug in `vpp43` itself. So if you don't see why the exception was raised, contact the current maintainers of PyVISA.

4.3 Legacy API

4.3.1 `legacy.visa` functions

`get_instruments_list`([*use_aliases*])
returns a list with all instruments that are known to the local VISA system. If you're lucky, these are all instruments connected with the computer. The boolean *use_aliases* is *True* by default, which means that the more human- friendly aliases like *"COM1"* instead of *"ASRL1"* are returned. With some VISA systems you can define your own aliases for each device, e.g. *"keithley617"* for *"GPIB0::15::INSTR"*. If *use_aliases* is *False*, only standard resource names are returned.

`instrument`(*resource_name*[, ***keyw*])
returns an instrument variable for the instrument given by *resource_name*. It saves you from calling one of the instrument classes directly by choosing the right one according to the type of the instrument. So you have *one* function to open *all* of your instruments.

The parameter *resource_name* may be any valid VISA instrument resource name, see section *VISA resource names*. In particular, you can use a name returned by `get_instruments_list()` above.

All further keyword arguments given to this function are passed to the class constructor of the respective instrument class. See section *General devices* for a table with all allowed keyword arguments and their meanings.

4.3.2 Module classes

General devices

class **`Instrument`**(*resource_name*[, ***keyw*])

represents an instrument, e.g. a measurement device. It is independent of a particular bus system, i.e. it may be a GPIB, serial, USB, or whatever instrument. However, it is not possible to perform bus-specific operations on instruments created by this class. For this, have a look at the specialised classes like `GpibInstrument` (section *Common properties of instrument variables*).

The parameter *resource_name* takes the same syntax as resource specifiers in VISA. Thus, it begins with the bus system followed by *"::"*, continues with the location of the device within the bus system, and ends with an optional *"::INSTR"*.

Possible keyword arguments are:

Keyword	Description
timeout	timeout in seconds for all device operations, see section *Timeouts*. Default: 5
chunk_size	Length of read data chunks in bytes, see section *Chunk length*. Default: 20kB
val-ues_format	Data format for lists of read values, see section *Reading binary data*. Default: *ascii*
term_char	termination characters, see section *Termination characters*. Default: *None*
send_end	whether to assert END after each write operation, see section *Termination characters*. Default: *True*
delay	delay in seconds after each write operation, see section *Termination characters*. Default: 0
lock	whether you want to have exclusive access to the device. Default: *VI_NO_LOCK*

For further information about the locking mechanism, see The VISA library implementation.

The class `Instrument` defines the following methods and attributes:

`Instrument.`**`write`**(*message*)

writes the string *message* to the instrument.

`Instrument.`**`read`**()

returns a string sent from the instrument to the computer.

`Instrument.`**`read_values`**([*format*])

returns a list of decimal values (floats) sent from the instrument to the computer. See section *A more complex example* above. The list may contain only one element or may be empty.

The optional *format* argument overrides the setting of *values_format*. For information about that, see section *Reading binary data*.

`Instrument.`**`ask`**(*message*)

sends the string *message* to the instrument and returns the answer string from the instrument.

`Instrument.`**`ask_for_values`**(*message*[, *format*])

sends the string *message* to the instrument and reads the answer as a list of values, just as *read_values()* does.

The optional *format* argument overrides the setting of *values_format*. For information about that, see section *Reading binary data*.

`Instrument.`**`clear`**`()`
resets the device. This operation is highly bus-dependent. I refer you to the original VISA documentation, which explains how this is achieved for VXI, GPIB, serial, etc.

`Instrument.`**`trigger`**`()`
sends a trigger signal to the instrument.

`Instrument.`**`read_raw`**`()`
returns a string sent from the instrument to the computer. In contrast to *read()*, no termination characters are checked or stripped. You get the pristine message.

`Instrument.`**`timeout`**
The timeout in seconds for each I/O operation. See section *Timeouts* for further information.

`Instrument.`**`term_chars`**
The termination characters for each read and write operation. See section *Termination characters* for further information.

`Instrument.`**`send_end`**
Whether or not to assert EOI (or something equivalent, depending on the interface type) after each write operation. See section *Termination characters* for further information.

`Instrument.`**`delay`**
Time in seconds to wait after each write operation. See section *Termination characters* for further information.

`Instrument.`**`values_format`**
The format for multi-value data sent from the instrument to the computer. See section *Reading binary data* for further information.

GPIB devices

class **`GpibInstrument`**`(`*gpib_identifier*[, *board_number*[, ***keyw*]]`)`
represents a GPIB instrument. If *gpib_identifier* is a string, it is interpreted as a VISA resource name (section *VISA resource names*). If it is a number, it denotes the device number at the GPIB bus.

The optional *board_number* defaults to zero. If you have more that one GPIB bus system attached to the computer, you can select the bus with this parameter.

The keyword arguments are interpreted the same as with the class `Instrument`.

Note: Since this class is derived from the class `Instrument`, please refer to section *General devices* for the basic operations. `GpibInstrument` can do everything that `Instrument` can do, so it simply extends the original class with GPIB-specific operations.

The class `GpibInstrument` defines the following methods:

`GpibInstrument.`**`wait_for_srq`**`(`[*timeout*]`)`
waits 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.

The *timeout* argument, given in seconds, denotes the maximal waiting time. The default value is 25 (seconds). If you pass *None* for the timeout, this method waits forever if no SRQ arrives.

class **`Gpib`**`(`[*board_number*]`)`
represents a GPIB board. Although most setups have at most one GPIB interface card or USB-GPIB device (with board number 0), theoretically you may have more. Be that as it may, for board-level operations, i.e. operations that affect the whole bus with all connected devices, you must create an instance of this class.

The optional GPIB board number *board_number* defaults to 0.

The class `Gpib` defines the following method:

`Gpib.`**`send_ifc`**`()`

pulses the interface clear line (IFC) for at least 0.1 seconds.

Note: You needn't store the board instance in a variable. Instead, you may send an IFC signal just by saying *Gpib().send_ifc()*.

Serial devices

Please note that "serial instrument" means only RS232 and parallel port instruments, i.e. everything attached to COM and LPT. In particular, it does not include USB instruments. For USB you have to use `Instrument` instead.

class **`SerialInstrument`**`(`*resource_name*[, ***keyw*]`)`

represents a serial instrument. *resource_name* is the VISA resource name, see section *VISA resource names*. The general keyword arguments are interpreted the same as with the class `Instrument`. The only difference is the default value for *term_chars*: For serial instruments, *CR* (carriage return) is used to terminate readings and writings.

Note: Since this class is derived from the class `Instrument`, please refer to section *General devices* for all operations. `SerialInstrument` can do everything that `Instrument` can do.

The class `SerialInstrument` defines the following additional properties. Note that all properties can also be given as keyword arguments when calling the class constructor or `instrument()`.

`SerialInstrument.`**`baud_rate`**

The communication speed in baud. The default value is 9600.

`SerialInstrument.`**`data_bits`**

Number of data bits contained in each frame. Its value must be from 5 to 8. The default is 8.

`SerialInstrument.`**`stop_bits`**

Number of stop bits contained in each frame. Possible values are 1, 1.5, and 2. The default is 1.

`SerialInstrument.`**`parity`**

The parity used with every frame transmitted and received. Possible values are:

Value	Description
no_parity	no parity bit is used
odd_parity	the parity bit causes odd parity
even_parity	the parity bit causes even parity
mark_parity	the parity bit exists but it's always 1
space_parity	the parity bit exists but it's always 0

The default value is *no_parity*.

`SerialInstrument.`**`end_input`**

This determines the method used to terminate read operations. Possible values are:

Value	Description
last_bit_end_input	read will terminate as soon as a character arrives with its last data bit set
term_chars_end_input	read will terminate as soon as the last character of *term_chars* is received

The default value is *term_chars_end_input*.

4.3.3 Common properties of instrument variables

4.3.4 Timeouts

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.

For all PyVISA objects, a timeout is set with

```
my_device.timeout = 25
```

Here, *my_device* may be a device, an interface or whatever, and its timeout is set to 25 seconds. Floating-point values are allowed. If you set it to zero, all operations must succeed instantaneously. You must not set it to *None*. Instead, if you want to remove the timeout, just say

```
del my_device.timeout
```

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

The default timeout is 5 seconds, but you can change it when creating the device object:

```
my_instrument = instrument("ASRL1", timeout = 8)
```

This creates the object variable *my_instrument* and sets its timeout to 8 seconds. In this context, a timeout value of *None* is allowed, which removes the timeout for this device.

Note that your local VISA library may round up this value heavily. I experienced this effect with my National Instruments VISA implementation, which rounds off to 0, 1, 3 and 10 seconds.

4.3.5 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.

4.3.6 Reading binary data

Some instruments allow for sending the measured data in binary form. This has the advantage that the data transfer is much smaller and takes less time. PyVISA currently supports three forms of transfers:

ascii This is the default mode. It assumes a normal string with comma- or whitespace-separated values.

single The values are expected as a binary sequence of IEEE floating point values with single precision (i.e. four bytes each).

double The same as **single**, but with values of double precision (eight bytes each).

You can set the form of transfer with the property *values_format*, either with the generation of the object,

```
my_instrument = instrument("GPIB::12", values_format = single)
```

or later by setting the property directly:

```
my_instrument.values_format = single
```

Setting this option affects the methods *read_values()* and *ask_for_values()*. In particular, you must assure separately that the device actually sends in this format. In some cases it may be necessary to set the *byte order*, also known as *endianness*. PyVISA assumes little-endian as default. Some instruments call this "swapped" byte order. However, there is also big-endian byte order. In this case you have to append | *big_endian* to your values format:

```
my_instrument = instrument("GPIB::12", values_format = single | big_endian)
```

Example

In order to demonstrate how easy reading binary data can be, remember our example from section *A more complex example*. You just have to append the lines

```
keithley.write("format:data sreal")
keithley.values_format = single
```

to the initialisation commands, and all measurement data will be transmitted as binary. You will only notice the increased speed, as PyVISA converts it into the same list of values as before.

4.3.7 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.term_chars = CR
```

Alternatively you can give it when creating your instrument object:

```
my_instrument = instrument("GPIB::10", term_chars = CR)
```

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

Well, the real default is not *""* (the empty string) but *None*. There is a subtle difference: *""* really means the termination characters are not used at all, neither for read nor for write operations. In contrast, *None* means that every write operation is implicitly terminated with *CR+LF* . This works well with most instruments.

All CRs and LFs are stripped from the end of a read string, no matter how *term_chars* is set.

The termination characters sequence is an ordinary string. *CR* and *LF* are just string constants that allow readable access to *"\r"* and *"\n"*. Therefore, instead of *CR+LF*, you can also write *"\r\n"*, whichever you like more.

delay and *send_end*

There are two further options related to message termination, namely *send_end* and *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 *delay* is the time in seconds to wait after each write operation. So you could write:

```
my_instrument = instrument("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.

4.3.8 Mixing with direct VISA commands

You can mix the high-level object-oriented approach described in this document with middle-level VISA function calls in module `vpp43` as described in The VISA library implementation which is also part of the PyVISA package. By doing so, you have full control of your devices. I recommend to import the VISA functions with:

```
from pyvisa import vpp43
```

Then you can use them with *vpp43.function_name(...)*.

The VISA functions need to know what session you are referring to. PyVISA opens exactly one session for each instrument or interface and stores its session handle in the instance attribute `vi`. For example, these two lines are equivalent:

```
my_instrument.clear()
vpp43.clear(my_instrument.vi)
```

In case you need the session handle for the default resource manager, it's stored in `resource_manager.session`:

```
from visa import *
from pyvisa import vpp43
my_instrument_handle = vpp43.open(resource_manager.session, "GPIB::14",
                                  VI_EXCLUSIVE_LOCK)
```

Setting the VISA library in the program

You can also set the path to your VISA library at the beginning of your program. Just start the program with

```
from pyvisa.vpp43 import visa_library
visa_library.load_library("/usr/lib/libvisa.so.7")
from visa import *
...
```

Keep in mind that the backslashes of Windows paths must be properly escaped, or the path must be preceeded by *r*:

```
from pyvisa.vpp43 import visa_library
visa_library.load_library(r"c:\WINNT\system32\visa32.dll")
from visa import *
...
```

4.3.9 `legacy.vpp43` functions

Please note that all descriptions given in this reference serve mostly as reminders. For real descriptions consult a VISA specification or NI-VISA Programmer Reference Manual. However, whenever there are PyVISA-specific semantics, they are listed here, too.

assert_interrupt_signal

Asserts the specified device interrupt or signal.

Call assert_interrupt_signal(vi, mode, status_id)

VISA name viAssertIntrSignal

Parameters

vi [integer] Unique logical identifier to a session.

mode [integer] This specifies how to assert the interrupt.

status_id [integer] This is the status value to be presented during an interrupt acknowledge cycle.

Return values None.

assert_trigger

Assert software or hardware trigger.

Call assert_trigger(vi, protocol)

VISA name viAssertTrigger

Parameters

vi [integer] Unique logical identifier to a session.

protocol [integer] Trigger protocol to use during assertion. Valid values are: `VI_TRIG_PROT_DEFAULT`, `VI_TRIG_PROT_ON`, `VI_TRIG_PROT_OFF`, and `VI_TRIG_PROT_SYNC`.

Return values None.

assert_utility_signal

Asserts the specified utility bus signal.

Call assert_utility_signal(vi, line)

VISA name viAssertUtilSignal

Parameters

vi [integer] Unique logical identifier to a session.

line [integer] Specifies the utility bus signal to assert.

Return values None.

buffer_read

Similar to read, except that the operation uses the formatted I/O read buffer for holding data read from the device.

Call buffer = buffer_read(vi, count)

VISA name viBufRead

Parameters

vi [integer] Unique logical identifier to a session.

count [integer] Maximal number of bytes to be read.

Return values

buffer [string] The buffer with the received data from device.

buffer_write

Similar to write, except the data is written to the formatted I/O write buffer rather than directly to the device.

Call return_count = buffer_write(vi, buffer)

VISA name viBufWrite

Parameters

vi [integer] Unique logical identifier to a session.

buffer [string] The data block to be sent to device.

Return values

return_count [integer] The number of bytes actually transferred.

clear

Clear a device.

Call clear(vi)

VISA name viClear

Parameters

vi [integer] Unique logical identifier to a session.

Return values None.

close

Close the specified session, event, or find list.

Call close(vi)

VISA name viClose

Parameters

vi [integer, ViEvent, or ViFindList] Unique logical identifier to a session, event, or find list.

Return values None.

disable_event

Disable notification of an event type by the specified mechanisms.

Call disable_event(vi, event_type, mechanism)

VISA name viDisableEvent

Parameters

vi [integer] Unique logical identifier to a session.

event_type [integer] Logical event identifier.

mechanism [integer] Specifies event handling mechanisms to be disabled. The queuing mechanism is disabled by specifying `VI_QUEUE`, and the callback mechanism is disabled by specifying `VI_HNDLR` or `VI_SUSPEND_HNDLR`. It is possible to disable both mechanisms simultaneously by specifying `VI_ALL_MECH`.

Return values None.

discard_events

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

Call discard_events(vi, event_type, mechanism)

VISA name viDiscardEvents

Parameters

vi [integer] Unique logical identifier to a session.

event_type [integer] Logical event identifier.

mechanism [integer] Specifies the mechanisms for which the events are to be discarded. The `VI_QUEUE` value is specified for the queuing mechanism and the `VI_SUSPEND_HNDLR` value is specified for the pending events in the callback mechanism. It is possible to specify both mechanisms simultaneously by specifying `VI_ALL_MECH`.

Return values None.

enable_event

Enable notification of a specified event.

Call enable_event(vi, event_type, mechanism, context)

VISA name viEnableEvent

Parameters

vi [integer] Unique logical identifier to a session.

event_type [integer] Logical event identifier.

mechanism [integer] Specifies event handling mechanisms to be enabled. The queuing mechanism is enabled by specifying `VI_QUEUE`, and the callback mechanism is enabled by specifying `VI_HNDLR` or `VI_SUSPEND_HNDLR`. It is possible to enable both mechanisms simultaneously by specifying bit-wise “or” of `VI_QUEUE` and one of the two mode values for the callback mechanism.

context [integer][optional] According to the VISA specification, this must be `Vi_NULL` always. (This is also the default value, of course.)

Return values None.

find_next

Call instrument_description = find_next(find_list)

VISA name viFindNext

Parameters

find_list [ViFindList] Describes a find list. This parameter must be created by find_resources.

Return values

instrument_description [string] Returns a string identifying the location of a device. Strings can then be passed to open to establish a session to the given device.

find_resources

Call find_list, return_counter, instrument_description = find_resources(session, regular_expression)

VISA name viFindRsrc

Parameters

session [integer] Resource Manager session (should always be the Default Resource Manager for VISA returned from open_default_resource_manager).

regular_expression [integer] This is a regular expression followed by an optional logical expression.

Return values

find_list [ViFindList] Returns a handle identifying this search session. This handle will be used as an input in find_next.

return_counter [integer] Number of matches.

instrument_description [string] Returns a string identifying the location of a device. Strings can then be passed to open to establish a session to the given device.

flush

Manually flush the specified buffers associated with formatted I/O operations and/or serial communication.

Call flush(vi, mask)

VISA name viFlush

Parameters

vi [integer] Unique logical identifier to a session.

mask [integer] Specifies the action to be taken with flushing the buffer.

Return values None.

get_attribute

Retrieve the state of an attribute.

Call attribute_state = get_attribute(vi, attribute)

VISA name viGetAttribute

Parameters

vi [integer, ViEvent, or ViFindList] Unique logical identifier to a session.

attribute [integer] Session, event, or find list attribute for which the state query is made.

Return values

attribute_state [integer, string, or list of integers] The state of the queried attribute for a specified resource.

gpib_command

Write GPIB command bytes on the bus.

Call return_count = gpib_command(vi, buffer)

VISA name viGpibCommand

Parameters

vi [integer] Unique logical identifier to a session.

buffer [string] Buffer containing valid GPIB commands.

Return values

return_count [integer] Number of bytes actually transferred.

gpib_control_atn

Controls the state of the GPIB ATN interface line, and optionally the active controller state of the local interface board.

Call gpib_control_atn(vi, mode)

VISA name viGpibControlATN

Parameters

vi [integer] Unique logical identifier to a session.

mode [integer] Specifies the state of the ATN line and optionally the local active controller state. See the Description section for actual values.

Return values None.

gpib_control_ren

Controls the state of the GPIB REN interface line, and optionally the remote/local state of the device.

Call gpib_control_ren(vi, mode)

VISA name viGpibControlREN

Parameters

vi [integer] Unique logical identifier to a session.

mode [integer] Specifies the state of the REN line and optionally the device remote/local state. See the Description section for actual values.

Return values None.

gpib_pass_control

Tell the GPIB device at the specified address to become controller in charge (CIC).

Call gpib_pass_control(vi, primary_address, secondary_address)

VISA name viGpibPassControl

Parameters

vi [integer] Unique logical identifier to a session.

primary_address [integer] Primary address of the GPIB device to which you want to pass control.

secondary_address [integer] Secondary address of the targeted GPIB device. If the targeted device does not have a secondary address, this parameter should contain the value `VI_NO_SEC_ADDR`.

Return values None.

gpib_send_ifc

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

Call gpib_send_ifc(vi)

VISA name viGpibSendIFC

Parameters

vi [integer] Unique logical identifier to a session.

Return values None.

in_8, in_16, in_32

Read in an 8-bit, 16-bit, or 32-bit value from the specified memory space and offset.

Call

value_8 = in_8(vi, space, offset)
value_16 = in_16(vi, space, offset)
value_32 = in_32(vi, space, offset)

VISA name viIn8, viIn16, viIn32

Parameters

vi [integer] Unique logical identifier to a session.

space [integer] Specifies the address space.

offset [integer] Offset in bytes of the address or register from which to read.

Return values

value_8*, *value_16*, *value_32 [integer] Data read from bus (8 bits for *in_8*, 16 bits for *in_16*, and 32 bits for *in_32*).

install_handler

Install handlers for event callbacks. A handler must have the following signature:

```
def event_handler(vi, event_type, context, user_handle):
    ...
```

Its parameters mean the following:

vi [integer] Unique logical identifier to a session.

event_type [ViEvent] Logical event identifier. With `event_type.value` you get its value as an integer.

context [ViEvent] A handle specifying the unique occurrence of an event.

user_handle [ctypes pointer type] A *pointer* to the user handle in ctypes form. See below at “Return values” for how to use it, however, you have to substitute `user_handle.contents` for `converted_user_handle` in the explanation.

Call converted_user_handle = install_handler(vi, event_type, handler, user_handle)

VISA name viInstallHandler

Parameters

vi [integer] Unique logical identifier to a session.

event_type [integer] Logical event identifier.

handler [callable] Interpreted as a valid reference to a handler to be installed by a client application.

user_handle [`None`, float, integer, string, or list of floats or integers][optional] A value specified by an application that can be used for identifying handlers uniquely for an event type. It defaults to `None`.

Return values

converted_user_handle [ctypes type] An object representing the user_handle. Use it to communicate with your handler. If your user_handle was a list, you get its elements as usual with `converted_user_handle[index]`. You can even convert it to a list with `list(converted_user_handle)` (however, this yields a copy).

For strings, use `converted_user_handle.value` if it’s supposed to be interpreted as a null-terminated string, or `converted_user_handle.raw` if you want to see *all* bytes. You can also write to both expressions, however, slicing is only possible for reading.

For simple types, you can say `converted_user_handle.value` (read and write).

Attention: You must assure that you never write values to converted_user_data which are longer (in bytes) than the initial values. So be careful not to write a string longer than the original one, nor a longer list. You’d be alerted by exceptions, though.

lock

Establish an access mode to the specified resource.

Call access_key = lock(vi, lock_type, timeout, requested_key)

VISA name viLock

Parameters

vi [integer] Unique logical identifier to a session.

lock_type [integer] Specifies the type of lock requested, which can be either `VI_EXCLUSIVE_LOCK` or `VI_SHARED_LOCK`.

timeout [integer] Absolute time period in milliseconds that a resource waits to get unlocked by the locking session before returning this operation with an error.

requested_key [ctypes string][optional] This parameter is not used if *lock_type* is `VI_EXCLUSIVE_LOCK` (exclusive locks). When trying to lock the resource as `VI_SHARED_LOCK` (shared), you can either omit it so that VISA generates an *access_key* for the session, or you can suggest an *access_key* to use for the shared lock.

Return values

access_key [ctypes string][optional] This value is `None` if *lock_type* is `VI_EXCLUSIVE_LOCK` (exclusive locks). When trying to lock the resource as `VI_SHARED_LOCK` (shared), the function returns a unique access key for the lock if the operation succeeds. This *access_key* can then be passed to other sessions to share the lock.

map_address

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

Call address = map_address(vi, map_space, map_base, map_size, access, suggested)

VISA name viMapAddress

Parameters

vi [integer] Unique logical identifier to a session.

map_space [integer] Specifies the address space to map.

map_base [ViBusAddress] Offset in bytes of the memory to be mapped.

map_size [integer] Amount of memory to map in bytes.

access [integer][optional] Must be `VI_FALSE`.

suggested [integer][optional] If not `VI_NULL` (the default), the operating system attempts to map the memory to the address specified in suggested. 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 suggested.

Return values

address [ViAddr] Address in your process space where the memory was mapped.

map_trigger

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

Call map_trigger(vi, trigger_source, trigger_destination, mode)

VISA name viMapTrigger

Parameters

vi [integer] Unique logical identifier to a session.

trigger_source [integer] Source line from which to map.

trigger_destination [integer] Destination line to which to map.

mode [integer] Specifies the trigger mapping mode. This should always be VI_NULL.

Return values None.

memory_allocation

Allocate memory from a device's memory region.

Call memory_allocation(vi, size)

VISA name viMemAlloc

Parameters

vi [integer] Unique logical identifier to a session.

size [integer] Specifies the size of the allocation.

Return values

offset [ViBusAddress] Returns the offset of the allocated device memory.

memory_free

Free memory previously allocated using memory_allocation.

Call memory_free(vi, offset)

VISA name viMemFree

Parameters

vi [integer] Unique logical identifier to a session.

offset [ViBusAddress] Specifies the memory previously allocated with memory_allocation.

Return values None.

move

Move a block of data.

Call move(vi, source_space, source_offset, source_width, destination_space, destination_offset, destination_width, length)

VISA name viMove

Parameters

vi [integer] Unique logical identifier to a session.

source_space [integer] Specifies the address space of the source.

source_offset [integer] Offset in bytes of the starting address or register from which to read.

source_width [integer] Specifies the data width of the source.

destination_space [integer] Specifies the address space of the destination.

destination_offset [integer] Offset in bytes of the starting address or register to which to write.

destination_width [integer] Specifies the data width of the destination.

length [integer] Number of elements to transfer, where the data width of the elements to transfer is identical to source data width.

Return values None.

move_asynchronously

Move a block of data asynchronously.

Call job_id = move_asynchronously(vi, source_space, source_offset, source_width, destination_space, destination_offset, destination_width, length)

VISA name viMoveAsync

Parameters

vi [integer] Unique logical identifier to a session.

source_space [integer] Specifies the address space of the source.

source_offset [integer] Offset in bytes of the starting address or register from which to read.

source_width [integer] Specifies the data width of the source.

destination_space [integer] Specifies the address space of the destination.

destination_offset [integer] Offset in bytes of the starting address or register to which to write.

destination_width [integer] Specifies the data width of the destination.

length [integer] Number of elements to transfer, where the data width of the elements to transfer is identical to source data width.

Return values

job_id [ViJobId] The job identifier of this asynchronous move operation. Each time an asynchronous move operation is called, it is assigned a unique job identifier.

move_in_8, move_in_16, move_in_32

Move a block of data from the specified address space and offset to local memory in increments of 8, 16, or 32 bits.

Call

buffer_8 = move_in_8(vi, space, offset, length)
buffer_16 = move_in_16(vi, space, offset, length)
buffer_32 = move_in_32(vi, space, offset, length)

VISA name viMoveIn8, viMoveIn16, viMoveIn32

Parameters

vi [integer] Unique logical identifier to a session.

space [integer] Specifies the address space.

offset [ViBusAddress] Offset in bytes of the starting address or register from which to read.

length [integer] Number of elements to transfer, where the data width of the elements to transfer is identical to data width (8, 16, or 32 bits).

Return values

buffer_8, buffer_16, buffer_32 [list of integers] Data read from bus as a Python list of values.

move_out_8, move_out_16, move_out_32

Move a block of data from local memory to the specified address space and offset in increments of 8, 16, or 32 bits.

Call

move_out_8(vi, space, offset, length, buffer_8)
move_out_16(vi, space, offset, length, buffer_16)
move_out_32(vi, space, offset, length, buffer_32)

VISA name viMoveOut8, viMoveOut16, viMoveOut32

Parameters

vi [integer] Unique logical identifier to a session.

space [integer] Specifies the address space.

offset [ViBusAddress] Offset in bytes of the starting address or register from which to write.

length [integer] Number of elements to transfer, where the data width of the elements to transfer is identical to data width (8, 16, or 32 bits).

buffer_8*, *buffer_16*, *buffer_32 [sequence of integers] Data to write to bus. This may be a list or a tuple, however in any case in must contain integers.

Return values None.

open

Open a session to the specified device.

Call vi = open(session, resource_name, access_mode, open_timeout)

VISA name viOpen

Parameters

session [integer] Resource Manager session (should always be the Default Resource Manager for VISA returned from open_default_resource_manager).

resource_name [string] Unique symbolic name of a resource.

access_mode [integer][optional] Defaults to `VI_NO_LOCK`. Specifies the modes by which the resource is to be accessed. The value `VI_EXCLUSIVE_LOCK` is used to acquire an exclusive lock immediately upon opening a session; if a lock cannot be acquired, the session is closed and an error is returned. The value `VI_LOAD_CONFIG` is used to configure attributes to values specified by some external configuration utility; if this value is not used, the session uses the default values provided by this specification. Multiple access modes can be used simultaneously by specifying a "bitwise OR" of the above values.

open_timeout [integer][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; otherwise, this parameter is ignored. Defaults to `VI_TMO_IMMEDIATE`.

Return values

vi [integer] Unique logical identifier reference to a session.

open_default_resource_manager

Return a session to the Default Resource Manager resource.

Call session = open_default_resource_manager()

VISA name viOpenDefaultRM

Parameters None.

Return values

session [integer] Unique logical identifier to a Default Resource Manager session.

get_default_resource_manager

This is a deprecated alias for open_default_resource_manager.

out_8, out_16, out_32

Call

out_8(vi, space, offset, value_8)
out_16(vi, space, offset, value_16)
out_32(vi, space, offset, value_32)

VISA name viOut8, viOut16, viOut32

Parameters

vi [integer] Unique logical identifier to a session.

space [integer] Specifies the address space.

offset [integer] Offset in bytes of the address or register to which to write.

***value_8*, *value_16*, *value_32*: integer** Data to write to bus (8 bits for out_8, 16 bits for out_16, and 32 bits for out_32).

Return values None.

parse_resource

Parse a resource string to get the interface information.

Call interface_type, interface_board_number = parse_resource(session, resource_name)

VISA name viParseRsrc

Parameters

session [integer] Resource Manager session (should always be the Default Resource Manager for VISA returned from open_default_resource_manager).

resource_name [string] Unique symbolic name of a resource.

Return values

interface_type [integer] Interface type of the given resource string.

interface_board_number [integer] Board number of the interface of the given resource string.

parse_resource_extended

Parse a resource string to get extended interface information.

Attention: Calling this function may raise an `AttributeError` because some older VISA implementation don't have the function `viParseRsrcEx`.

Call interface_type, interface_board_number, resource_class, unaliased_expanded_resource_name, alias_if_exists = parse_resource_extended(session, resource_name)

VISA name viParseRsrcEx

Parameters

session [integer] Resource Manager session (should always be the Default Resource Manager for VISA returned from open_default_resource_manager).

resource_name [string] Unique symbolic name of a resource.

Return values

interface_type [integer] Interface type of the given resource string.

interface_board_number [integer] Board number of the interface of the given resource string.

resource_class [string] Specifies the resource class (for example "INSTR") of the given resource string.

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

alias_if_exists [string] Specifies the user-defined alias for the given resource string, if a VISA implementation allows aliases and an alias exists for the given resource string. If not, this is `None`.

peek_8, peek_16, peek_32

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

Call

value_8 = peek_8(vi, address)
value_16 = peek_16(vi, address)
value_32 = peek_32(vi, address)

VISA name viPeek8, viPeek16, viPeek32

Parameters

vi [integer] Unique logical identifier to a session.

address [ViAddr] Specifies the source address to read the value.

Return values

value_8, value_16, value_32 [integer] Data read from bus (8 bits for peek_8, 16 bits for peek_16, and 32 bits for peek_32).

poke_8, poke_16, poke_32

Write an 8-bit, 16-bit, or 32-bit value to the specified address.

Call

poke_8(vi, address, value_8)
poke_16(vi, address, value_16)
poke_32(vi, address, value_32)

VISA name vipoke_8

Parameters

vi [integer] Unique logical identifier to a session.

address [integer] Specifies the destination address to store the value.

value_8*, *value_16*, *value_32 [integer] Data to write to bus (8 bits for poke_8, 16 bits for poke_16, and 32 bits for poke_32).

Return values None.

printf

Convert, format, and send the parameters `...` to the device as specified by the format string.

> **Warning:** The current implementation only supports the following C data types: `long`, `double` and `char*` (strings). Thus, you can only use these three data types in format strings for printf, scanf and the like.

Call printf(vi, write_format, ...)

VISA name viPrintf

Parameters

vi [integer] Unique logical identifier to a session.

write_format [string] String describing the format for arguments.

... [integers, floats, or strings] Arguments sent to the device according to *write_format*.

Return values None.

queryf

Perform a formatted write and read through a single operation invocation.

> **Warning:** The current implementation only supports the following C data types: `long`, `double` and `char*` (strings). Thus, you can only use these three data types in format strings for printf, scanf and the like.

Call value1, value2, ... = queryf(vi, write_format, read_format, (...), ..., maximal_string_length = 1024)

VISA name viQueryf

Parameters

vi [integer] Unique logical identifier to a session.

write_format [string] String describing the format for arguments.

read_format [string] String describing the format for arguments.

(...) [tuple of integers, floats, or strings] Arguments sent to the device according to *write_format*. May be `None`.

... [integers, floats, or strings] Arguments to be read from the device according to *read_format*. It's totally insignificant which values they have, they serve just as a cheap way to determine what types are to be expected. So actually this argument list shouldn't be necessary, but with the current implementation, it is, sorry.

These arguments may be (however needn't be) the same names used for storing the result values. Alternatively, you can give literals.

maximal_string_length [integer][keyword argument] The maximal length assumed for string result arguments. Note that string results must *never* exceed this length. It defaults to 1024.

Return values

***value1*, *value2*, ...** [integers, floats, or strings] Arguments read from the device according to *read_format*. Of course, this must be the same sequence (as far as data types are concerned) as the given argument list ... above.

read

Read data from device synchronously.

Call buffer = read(vi, count)

VISA name viRead

Parameters

vi [integer] Unique logical identifier to a session.

count [integer] Maximal number of bytes to be read.

Return values

buffer [string] Represents the buffer with the received data from device.

read_asynchronously

Read data from device asynchronously.

Call buffer, job_id = read_asynchronously(vi, count)

VISA name viReadAsync

Parameters

vi [integer] Unique logical identifier to a session.

count [integer] Maximal number of bytes to be read.

Return values

buffer [ctypes string buffer] Represents the buffer with the data received from device. It's not a native Python data type because it's filled in the background (i.e. asynchronously). After you assured that the reading is finished, you get its value with:

```
buffer.raw[:return_count]
```

You get `return_count` via the attribute `VI_ATTR_RET_COUNT`. See the VISA reference for further information.

job_id [ViJobId] Represents the location of a variable that will be set to the job identifier of this asynchronous read operation.

read_stb

Read a status byte of the service request.

Call status = read_stb(vi)

VISA name viReadSTB

Parameters

vi [integer] Unique logical identifier to a session.

Return values

status [integer] Service request status byte.

read_to_file

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

Call return_count = read_to_file(vi, filename, count)

VISA name viReadToFile

Parameters

vi [integer] Unique logical identifier to a session.

file_name [string] Name of file to which data will be written.

count [integer] Maximal number of bytes to be read.

Return values

return_count [integer] Number of bytes actually transferred.

scanf

Read, convert, and format data using the format specifier. Store the formatted data in the given optional parameters.

Warning: The current implementation only supports the following C data types: `long`, `double` and `char*` (strings). Thus, you can only use these three data types in format strings for printf, scanf and the like.

Call value1, value2, ... = scanf(vi, read_format, ..., maximal_string_length = 1024)

VISA name viScanf

Parameters

vi [integer] Unique logical identifier to a session.

read_format [string] String describing the format for arguments.

... [integers, floats, or strings] Arguments to be read from the device according to *read_format*. It's totally insignificant which values they have, they serve just as a cheap way to determine what types are to be expected. So actually this argument list shouldn't be necessary, but with the current implementation, it is, sorry.

These arguments may be (however needn't be) the same names used for storing the result values. Alternatively, you can give literals.

maximal_string_length [integer][keyword argument] The maximal length assumed for string result arguments. Note that string results must *never* exceed this length. It defaults to 1024.

Return values

***value1*, *value2*, ...** [integers, floats, or strings] Arguments read from the device according to *read_format*. Of course, this must be the same sequence (as far as data types are concerned) as the given argument list ... above.

set_attribute

Set the state of an attribute.

Call set_attribute(vi, attribute, attribute_state)

VISA name viSetAttribute

Parameters

vi [integer, ViEvent, or ViFindList] Unique logical identifier to a session.

attribute [integer] Session, event, or find list attribute for which the state is modified.

attribute_state [integer] The state of the attribute to be set for the specified resource. The interpretation of the individual attribute value is defined by the resource.

Return values None.

set_buffer

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

Call set_buffer(vi, mask, size)

VISA name viSetBuf

Parameters

vi [integer] Unique logical identifier to a session.

mask [integer] Specifies the type of buffer.

size [integer] The size to be set for the specified buffer(s).

Return values None.

sprintf

Same as printf, except the data is written to a user-specified buffer rather than the device.

Warning: The current implementation only supports the following C data types: `long`, `double` and `char*` (strings). Thus, you can only use these three data types in format strings for printf, scanf and the like.

Call buffer = sprintf(vi, write_format, ..., buffer_length = 1024)

VISA name viSPrintf

Parameters

vi [integer] Unique logical identifier to a session.

write_format [string] String describing the format for arguments.

... [integers, floats, or strings] Arguments sent to the buffer according to *write_format*.

buffer_length [integer][keyword argument] Length of the user-specified buffer in bytes. Defaults to 1024.

Return values

buffer [string] Buffer where the formatted data was written to.

sscanf

Same as scanf, except that the data is read from a user-specified buffer instead of a device.

Warning: The current implementation only supports the following C data types: `long`, `double` and `char*` (strings). Thus, you can only use these three data types in format strings for printf, scanf and the like.

Call value1, value2, ... = sscanf(vi, buffer, read_format, ..., maximal_string_length = 1024)

VISA name viSScanf

Parameters

vi [integer] Unique logical identifier to a session.

buffer [string] Buffer from which data is read and formatted.

read_format [string] String describing the format for arguments.

... [integers, floats, or strings] Arguments to be read from the device according to *read_format*. It's totally insignificant which values they have, they serve just as a cheap way to determine what types are to be expected. So actually this argument list shouldn't be necessary, but with the current implementation, it is, sorry.

These arguments may be (however needn't be) the same names used for storing the result values. Alternatively, you can give literals.

maximal_string_length [integer][keyword argument] The maximal length assumed for string result arguments. Note that string results must *never* exceed this length. It defaults to 1024.

Return values

value1, value2, ... [integers, floats, or strings] Arguments read from the device according to *read_format*. Of course, this must be the same sequence (as far as data types are concerned) as the given argument list ... above.

status_description

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

Call description = status_description(vi, status)

VISA name viStatusDesc

Parameters

vi [integer, ViEvent, or ViFindList] Unique logical identifier to a session.

status [integer] Status code to interpret.

Return values

description [string] The user-readable string interpretation of the status code passed to the operation.

terminate

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

Call terminate(vi, degree, job_id)

VISA name viTerminate

Parameters

vi [integer] Unique logical identifier to a session.

degree [integer] `VI_NULL`

job_id [ViJobId] Specifies an operation identifier.

Return values None.

uninstall_handler

Uninstall handlers for events.

Call uninstall_handler(vi, event_type, handler, user_handle)

VISA name viUninstallHandler

Parameters

vi [integer] Unique logical identifier to a session.

event_type [integer] Logical event identifier.

handler [callable] Interpreted as a valid reference to a handler to be uninstalled by a client application.

user_handle [ctypes type][optional] A value specified by an application that can be used for identifying handlers uniquely in a session for an event. It *must* be the object returned by install_handler. Consequently, it defaults to `None`.

Return values None.

unlock

Relinquish a lock for the specified resource.

Call unlock(vi)

VISA name viUnlock

Parameters

vi [integer] Unique logical identifier to a session.

Return values None.

unmap_address

Unmap memory space previously mapped by map_address.

Call unmap_address(vi)

VISA name viUnmapAddress

Parameters

vi [integer] Unique logical identifier to a session.

Return values None.

unmap_trigger

Undo a previous map from the specified trigger source line to the specified destination line.

Call unmap_trigger(vi, trigger_source, trigger_destination)

VISA name viUnmapTrigger

Parameters

vi [integer] Unique logical identifier to a session.

trigger_source [integer] Source line used in previous map.

trigger_destination [integer] Destination line used in previous map.

Return values None.

usb_control_in

Request arbitrary data from the USB device on the control port.

Call buffer = usb_control_in(vi, request_type_bitmap_field, request_id, request_value, index, length)

VISA name viUsbControlIn

Parameters

vi [integer] Unique logical identifier to a session.

request_type_bitmap_field [integer] Bitmap field for defining the USB control port request. The bitmap fields are as defined by the USB specification. The direction bit must be device-to-host.

request_id [integer] Request ID for this transfer. The meaning of this value depends on *request_type_bitmap_field*.

request_value [integer] Request value for this transfer.

index [integer] Specifies the interface or endpoint index number, depending on *request_type_bitmap_field*.

length [integer][optional] Number of data in bytes to request from the device during the Data stage. If this value is not given or 0, an empty string is returned.

Return values

buffer [string] Actual data received from the device during the Data stage. If *length* is not given or 0, an empty string is returned.

usb_control_out

Send arbitrary data to the USB device on the control port.

Call usb_control_out(vi, request_type_bitmap_field, request_id, request_value, index, buffer)

VISA name viUsbControlOut

Parameters

vi [integer] Unique logical identifier to a session.

request_type_bitmap_field [integer] Bitmap field for defining the USB control port request. The bitmap fields are as defined by the USB specification. The direction bit must be host-to-device.

request_id [integer] Request ID for this transfer. The meaning of this value depends on *request_type_bitmap_field*.

request_value [integer] Request value for this transfer.

index [integer] Specifies the interface or endpoint index number, depending on *request_type_bitmap_field*.

buffer [string][optional] Actual data to send to the device during the Data stage. If not given, nothing is sent.

Return values None.

vprintf, vqueryf, vscanf, vsprintf, vsscanf

These variants make no sense in Python, so I realised them as mere aliases (just drop the "v").

vxi_command_query

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

Call vxi_command_query(vi, mode, command)

VISA name viVxiCommandQuery

Parameters

vi [integer] Unique logical identifier to a session.

mode [integer] Specifies whether to issue a command and/or retrieve a response.

command [integer] The miscellaneous command to send.

Return values

response [integer] The response retrieved from the device. If the mode specifies just sending a command, this parameter may be `VI_NULL`.

wait_on_event

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

Call out_event_type, out_context = wait_on_event(vi, in_event_type, timeout)

VISA name viWaitOnEvent

Parameters

vi [integer] Unique logical identifier to a session.

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

timeout [integer] Absolute time period in milliseconds that the resource shall wait for a specified event to occur before returning the time elapsed error.

Return values

out_event_type [integer] Logical identifier of the event actually received.

out_context [ViEvent] A handle specifying the unique occurrence of an event.

write

Write data to device synchronously.

Call return_count = write(vi, buffer)

VISA name viWrite

Parameters

vi [integer] Unique logical identifier to a session.

buffer [string] Contains the data block to be sent to the device.

Return values

return_count [integer] The number of bytes actually transferred.

write_asynchronously

Write data to device asynchronously.

Call job_id = write_asynchronously(vi, buffer)

VISA name viWriteAsync

Parameters

vi [integer] Unique logical identifier to a session.

buffer [string] Contains the data block to be sent to the device.

Return values

job_id [ViJobId] The job identifier of this asynchronous write operation.

write_from_file

Take data from a file and write it out synchronously.

Call return_count = write_from_file(vi, filename, count)

VISA name viWriteFromFile

Parameters

vi [integer] Unique logical identifier to a session.

filename [string] Name of file from which data will be read.

count [integer] Maximal number of bytes to be written.

Return values

return_count [integer] Number of bytes actually transferred.

p

pyvisa.ctwrapper.functions, 42