Lantz Documentation

Release 0.1

Hernán E. Grecco

January 01, 2013

CONTENTS

Lantz is an automation and instrumentation toolkit with a clean, well-designed and consistent interface. It provides a core of commonly used functionalities for building applications that communicate with scientific instruments allowing rapid application prototyping, development and testing. Lantz benefits from Python's extensive library flexibility as a glue language to wrap existing drivers and DLLs. Lantz aims to provide a library of curated and well documented instruments drivers.

About

Lantz Philosophy and design principles.

Overview

A short tour of Lantz features.

FAQs

Frequently asked questions and their answers.

Community

Getting in touch with users and developers.

CHAPTER

ONE

LEARN

Tutorials

Step-by-step instructions to install, use and build drivers using Lantz.

Guides

Task oriented guides.

CHAPTER

TWO

MORE INFORMATION

Contributing
We need your brain.

Drivers
List of Lantz drivers.

API
Application programming interface reference.

CHAPTER

THREE

INDICES AND TABLES

genindex
Lists all sections and subsections.

modindex
All functions, classes, terms.

search
Search this documentation.

3.1 About

Lantz is an automation and instrumentation toolkit with a clean, well-designed and consistent API. It provides a core of commonly used functionalities enabling rapid prototyping and development of applications that communicate with scientific instruments.

Lantz provides out of the box a large set of instrument drivers. It additionally provides a complete set of base classes to write new drivers compatible with the toolkit. Lantz benefits from Python's flexibility as a glue language to wrap existing drivers and DLLs.

Lantz wraps common widgets for instrumentation, allowing to build responsive, event-driven user interfaces.

Lantz explains itself by contextually displaying documentation. Core functionalities, widgets and drivers adhere to a high documentation standard that allows the user to know exactly what the program is doing and the purpose of each parameter.

Lantz works well with Linux, Mac and Windows. It is written in Python and Qt4 for the user interface.

Lantz profits from Python's batteries-included philosophy and it's extensive library in many different fields from text parsing to database communication.

Lantz builds on the giant shoulders. By using state-of-the art libraries, it delivers tested robustness and performance.

Lantz speaks many languages. It is built with an international audience from the start thereby allowing translations to be made easily.

Lantz is free as in beer and free as in speech. You can view, modify and distribute the source code as long as you keep a reference to the original version and distribute your changes. It is distributed using the BSD License. See LICENSE for more details

3.2 Overview

A minimal script to control a function generator using Lantz might look like this:

```
from lantz import Q_

from lantz.drivers.aeroflex import A2023aSerial

fungen = A2023aSerial('COM1')
fungen.initialize()

print(fungen.idn)
fungen.frequency = Q_(20, 'MHz')
print(fungen.amplitude)
fungen.sweep()

fungen.finalize()
```

The code is basically self explanatory, and does not differ too much of what you would write if you write a driver from scratch. But there are a few important things going under the hood that makes Lantz useful for instrumentation. Let's take a look!

3.2.1 Logging

While building and running your program it is invaluable to monitor its state. Lantz gives to all your drivers automatic logging.

The default level is logging.INFO, but if you prepend the following lines to the previous example:

```
import logging
from lantz import log_to_screen
log_to_screen(logging.DEBUG)
```

You will see the instance initializing and how and when each property is accessed. Loggers are organized by default with the following naming convention:

```
lantz.<class name>.<instance name>
```

which for this case becomes:

```
lantz.A2023aSerial.A2023aSerial0
```

because no name was given. If you want to specify a name, do it at object creation:

```
fungen = A2023aSerial('COM1', name='white')
```

Separation into multiple loggers makes finding problems easier and enables fine grained control over log output.

By the way, if you are running your program from an IDE or you don't want to clutter your current terminal, you can log to a socket and view the log output in another window (even in another computer, but we leave this for latter). Open first another terminal and run:

```
$ lantzmonitor.py -l 1
```

(If you want a nicer user interface with filtering and searching capabilities, try LogView http://code.google.com/p/logview/)

To your python program, replace the logging lines by:

```
import logging
from lantz import log_to_socket
log_to_socket(logging.DEBUG)
```

When you run it, you will see the log appearing in the logging window.

By the way, *lantzmonitor* is more than log to screen dumper. Tailored for lantz, it can display instrument specific messages as well as an on-line summary indicating the current value for each property. Hopefully, you will never need to add a print statement in your program any more!

3.2.2 Timing

Basic statistics of instrument related function calls are kept to facilitate bottleneck identification. While this is not as powerful as python profiler, its much easier to use within your application. You can obtain the statistics for a particular operation using:

```
fungen.timing.stats('set_frequency')
```

This will return a named tuple with the following fields:

```
- last: Execution time of last set operation
- count: Number of times the setter was called
- mean: Mean execution time of all set operations
- std: Standard deviation of the execution time of all set operations
- min: Minimum execution time of all set operations
- max: Maximum execution time of all set operations
```

Similarly, you can obtain timing statistics of the getter calling:

```
fungen.timing.stats('get_frequency')
```

3.2.3 Cache

Setting and getting drivers properties always does it in the instrument. However, accessing the instrument is time consuming and many times you just want to a way to recall the last known value. Lantz properties carry their own cache, which can be accessed with the recall method:

```
>>> fungen.recall('amplitude')
20 V
```

You can also access multiple elements:

```
>>> fungen.recall(('amplitude', 'voltage'))
{'frequency': 20 MHz, 'amplitude': 20 V}
```

Using recall without arguments gets all defined feats

```
>>> fungen.recall()
{'frequency': 20 MHz, 'amplitude': 20 V, 'ac_mode': True }
```

3.2.4 Prevent unnecessary set

The internal cache also prevents unnecessary communication with the instrument:

```
>>> fungen.amplitude = 20 # The amplitude will be changed to 20
>>> fungen.amplitude = 20 # The amplitude is already 20, so this will be ignored.
```

If you are not sure that the current state of the instrument matches the cached value, you can force a setting change as will be described below.

3.2.5 Getting and setting multiple values in one line

You can use the refresh method to obtain multiple values from the instrument:

```
>>> print(fungen.refresh('amplitude')) # is equivalent to print(fungen.amplitude)
20 V

>>> print(fungen.refresh(('frequency', 'amplitude'))) # You can refresh multiple properties at once
{'frequency': 20 MHz, 'amplitude': 20 V}

>>> print(fungen.refresh()) # You can refresh all properties at once
{'frequency': 20 MHz, 'amplitude': 20 V, 'ac_mode': True }
```

The counterpart of refresh is the update method that allows you to set multiple values in a single line:

```
>>> fungen.update(ac_mode=True) # is equivalent to fungen.ac_mode = True

>>> fungen.update({'ac_mode': True})  # Can be also used with a dictionary

>>> fungen.update(ac_mode=True, amplitude=Q(42, 'V')) # if you want to set many, just do

>>> fungen.update({'ac_mode': True, 'amplitude': Q(42, 'V')}) # or this
```

The cache is what allows to Lantz to avoid unnecessary communication with the instrument. You can overrule this check using the update method:

```
>>> fungen.amplitude = Q(42, 'V')

>>> fungen.amplitude = Q(42, 'V') # No information is set to the instrument as is the value already s

>>> fungen.update(amplitude=Q(42, 'V'), force=True) # The force true argument ignores cache checking
```

This can be useful for example when the operator might change the settings using the manual controls.

3.2.6 Effortless asynchronous get and set

Lantz also provides out of the box asynchronous capabilities for all methods described before. For example:

```
>>> fungen.update_async({'ac_mode': True, 'amplitude': Q(42, 'V')})
>>> print('I am not blocked!')
```

will update *ac_mode* and *amplitude* without blocking, so the print statement is executed even if the update has not finished. This is useful when updating multiple independent instruments. The state of the operation can be verified using the returned `concurrent.futures.Future` object:

```
>>> result1 = fungen.update_async({'ac_mode': True, 'amplitude': Q(42, 'V')})
>>> result2 = another_fungen.update_async({'ac_mode': True, 'amplitude': Q(42, 'V')})
>>> while not result1.done() and not result2.done()
...     DoSomething()
```

Just like *update_async*, you can use *refresh_async* to obtain the value of one or more features. The result is again a `concurrent.futures.Future` object whose value can be queried using the result method `concurrent.futures.Future.result()`

```
>>> fut = obj.refresh_async('eggs')
>>> DoSomething()
>>> print(fut.result())
```

Async methods accept also a callback argument to define a method that will be used

Under the hood

Single thread for the instrument

3.2.7 Context manager

If you want to send a command to an instrument only once during a particular script, you might want to make use of the context manager syntax. In the following example, the driver will be created and initialized in the first line and finalized when the *with* clause finishes even when an unhandled exception is raised:

```
with A2023aSerial('COM1') as fungen:

    print(fungen.idn)
    fungen.frequency = Q_(20, 'MHz')
    print(fungen.amplitude)
    fungen.sweep()
```

3.2.8 Units

Instrumentation software need to deal with physical units, and therefore you need to deal with them. Keeping track of the units of each variable in time consuming and error prone, and derives into annoying naming practices such as *freq_in_KHz*. Lantz aims to reduce the burden of this by incorporating units using the Pint package. The Quantity object si abbreviated withing Lantz as *Q_* and can be imported from the root:

```
from lantz import Q_

mv = Q_(1, 'mV') # we define milivolt
value = 42 * mv # we can use the defined units like this
thesame = Q_(42, 'mv') # or like this
```

This makes the code a little more verbose but is worth the effort. The code is more explicit and less error prone. It also allows you to do thing like this:

```
from lantz import Q_

from lantz.drivers.example import OneFunGen as FunGen
# In OneFunGen, the amplitude of this function generator must be set in Volts.

with FunGen('COM1') as fungen:

    fungen.frequency = Q_(0.05, 'V')
```

Later you decide to change the function generator by a different one, with a different communication protocol:

```
from lantz import Q_

from lantz.drivers.example import AnotherFunGen as FunGen
# In AnotherFunGen, the amplitude of this function generator must be set in milivolts.

with FunGen('COM1') as fungen:

    fungen.frequency = Q_(0.05, 'V') # the value is converted from volts to mV inside the driver.
```

Apart from the import, nothing has changed. In a big code base this means that you can easily replace one instrument by another.

You might want to use the value obtained in one instrument to set another. Or you might want to use the same value in two different instruments without looking into their specific details:

```
from lantz import Q_

from lantz.drivers.example import FrequenceMeter
from lantz.drivers.aeroflex import A2023aSerial
from lantz.drivers.standford import SR844

with FrequenceMeter('COM1') as fmeter, \
     A2023aSerial('COM2') as fungen, \
     SR844('COM3') as lockin:

    freq = fmeter.frequency

    fungen.frequency = freq
    lockin.frequency = freq
```

In case you are not convinced, a small technical note:

Note: The MCO MIB has determined that the root cause for the loss of the MCO spacecraft was the failure to use metric units in the coding of a ground software file, "Small Forces," used in trajectory models. Specifically, thruster performance data in English units instead of metric units was used in the software application code titled SM_FORCES (small forces). The output from the SM_FORCES application code as required by a MSOP Project Software Interface Specification (SIS) was to be in metric units of Newtonseconds (N-s). Instead, the data was reported in English units of pound-seconds (lbf-s). The Angular Momentum Desaturation (AMD) file contained the output data from the SM_FORCES software. The SIS, which was not followed, defines both the format and units of the AMD file generated by ground-based computers. Subsequent processing of the data from AMD file by the navigation software algorithm therefore, underestimated the effect on the spacecraft trajectory by a factor of 4.45, which is the required conversion factor from force in pounds to Newtons. An erroneous trajectory was computed using this incorrect data.

Mars Climate Orbiter Mishap Investigation Phase I Report PDF

3.2.9 User interface

Providing a powerful GUI is an important aspect of developing an application for end user. Lantz aims to simplify the UI development by allowing you to correctly connect to *Lantz* Feats and Actions to widgets without any effort. For example, if you generate a GUI using Qt Designer:

```
# imports not shown

main = loadUi('connect_test.ui') # Load the GUI

with LantzSignalGeneratorTCP() as fungen: # Instantiate the instrument
```

```
    connect_driver(main, fungen) # All signals and slots are connected here!

    # Do something
```

Additionally it provides automatic generation of Test Panels, a very useful feature when you are building or debugging a new driver:

```
# imports not shown

with LantzSignalGeneratorTCP() as fungen: # Instantiate the instrument
    start_test_app(inst)                  # Create
```

and you get:


Lantz Driver Test Panel
LantzSignalGenerator LantzSignalGenerator0
Refresh
Update
Update on change
amplitude
0.00 volt
get
set
din
1
get
set
dout
1
get
set
frequency
1000.00 hertz
get
set
idn
ctionGenerator Serial #12345
get
set
offset
0.00 volt
get
set
output_enabled
get
set
waveform
sine
get
set
Actions:
self_test
Run

Check out the *Tutorials* to get started!

3.3 Tutorials

3.3.1 Installation guide

This guide describes Lantz requirements and provides platform specific installation guides. Examples are given for Python 3.2 installing all optional requirements as site-packages.

Requirements

Lantz core requires only Python 3.2+.

Optional requirements

Some lantz subpackages have other requirements which are listed below together with a small explanation of where are used. Short installation instructions are given, but we refer you to the package documentation for more information. For some packages, a link to the binary distribution is given. Specifi

- Colorama is used to colorize terminal output. It is optional when logging to screen and mandatory if you want to use *lantz-monitor*, the text-based log viewer.
- Sphinx is used generate the documentation. It is optional and only needed if you want to generate the documentation yourself.
- Docutils is used to transform the RST documentation to HTML which is then provided as tooltips in the GUI. It is optional. If not installed, unformatted documentation will be shown as tooltips. It will be already installed if you install Sphinx.
- pySerial it is to communicate via serial port. It is optional and only needed if you are using a driver that uses lantz.serial.
- Qt4 is used to generate the graphical user interfaces. Due to a license issue there are two python bindings for Qt: PyQt and PySide.
- NumPy is used by many drivers to perform numerical calculations.
- VISA National Instruments Library for communicating via GPIB, VXI, PXI, Serial, Ethernet, and/or USB interfaces
- *Linux*
- *OSX*
- *Windows*

Linux

Most linux distributions provide packages for Python 3.2, NumPy, PyQt (or PySide). There might be some other useful packages. For some distributions, you will find specific instructions below.

Ubuntu 12.04

```
$ sudo apt-get python3
$ sudo apt-get install python3-pkg-resources python3-pyqt4 python3-setuptools python3-sphinx
$ sudo apt-get install python3-numpy
```

and continue to to step 4 in OSX

Ubuntu 12.10

```
$ sudo apt-get python3
$ sudo apt-get install python3-pkg-resources python3-pyqt4 python3-setuptools python3-sphinx python3-
$ sudo apt-get install python3-numpy
```

and continue to to step 5 in OSX

openSUSE 12.2

```
$ sudo zypper install python3
$ sudo zypper install python3-pip python3-pyqt4 python3-Sphinx python-distutils-extra
$ sudo zypper install python3-numpy
```

and continue to to step 5 in OSX

OSX

1. Install Python 3.2
2. (optionally) Install PyQt, NumPy
3. (optionally) Install VISA
4. Open a terminal to install pip:

   ```
   $ curl http://python-distribute.org/distribute_setup.py | python3.2
   $ curl https://raw.github.com/pypa/pip/master/contrib/get-pip.py | python3.2
   ```

5. Using pip, install Lantz and its dependencies other optional dependencies:

   ```
   $ pip-3.2 install -r https://raw.github.com/hgrecco/lantz/master/requirements-full.txt
   ```

Windows

Note: We provide a simple script to run all the steps provided below. Download get-lantz to the folder in which you want to create the virtual environment. The run the script using a 32 bit version of Python 3.2+.

In some of the steps, an installer application will pop-up. Just select all default options.

As the script will download and install only necessary packages, it does not need a clean Python to start.

Install Python, NumPy binaries, PyQt binaries (or *PySide binaries*), VISA.

Download and run with Python 3.2:

```
- http://python-distribute.org/distribute_setup.py
- https://raw.github.com/pypa/pip/master/contrib/get-pip.py
```

In the command prompt install using pip all other optional dependencies:

```
$ C:\Python3.2\Scripts\pip install -r https://raw.github.com/hgrecco/lantz/master/requirements-full.t
```

If the driver from your instrument is available, you can start to use it right away. Learn how in the next part of the tutorial: *Using lantz drivers*.

3.3.2 Using lantz drivers

In this part of the tutorial, you will learn how to use Lantz drivers to control an instrument. Lantz is shipped with drivers for common laboratory instruments. Each instrument has different capabilities, and these reflect in the drivers being different. However, all Lantz drivers share a common structure and learning about it allows you to use them in a more efficient way.

Following a tutorial about using a driver to communicate with an instrument that you do not have is not much fun. That's why we have created a virtual version of this instrument. From the command line, run the following command:

```
$ lantz-sim fungen tcp
```

Note: If you are using Windows, it is likely that *lantz-sim* script is not be in the path. You will have to change directory to *C:\Python32\Scripts* or something similar.

This will start an application (i.e. your instrument) that listens for incoming TCP packages (commands) on port 5678 from *localhost*. In the screen you will see the commands received and sent by the instrument.

Your program and the instrument will communicate by exchanging text commands via TCP. But having a Lantz driver already built for your particular instrument releases you for the burden of sending and receiving the messages. Let's start by finding the driver. Lantz drivers are organized inside packages, each package named after the manufacturer. So the *Coherent Argon Laser Innova* 300C driver is in *lantz.drivers.coherent* under the name *ArgonInnova300C*. We follow Python style guide (PEP8) to name packages and modules (lowercase) and classes (CamelCase).

Make a new folder for your project and create inside a python script named *test_fungen.py*. Copy the following code inside the file:

```
from lantz.drivers.examples import LantzSignalGeneratorTCP

inst = LantzSignalGeneratorTCP('localhost', 5678)
inst.initialize()
print(inst.idn)
inst.finalize()
```

Let's look at the code line-by-line. First we import the class into our script:

```
from lantz.drivers.examples import LantzSignalGeneratorTCP
```

The driver for our simulated device is under the company *examples* and is named *LantzSignalGeneratorTCP*. Then we create an instance of the class, setting the address to localhost and port to 5678:

```
inst = LantzSignalGeneratorTCP('localhost', 5678)
```

This does not connects to the device. To do so, you call the *initialize* method:

```
inst.initialize()
```

All Lantz drivers have an *initialize* method. Drivers that communicate through a port (e.g. a Serial port) will open the port in this call. Then we query the instrument for it's identification and we print it:

```
print(inst.idn)
```

At the end, we call the *finalize* method to clean up all resources (e.g. close ports):

```
inst.finalize()
```

Just like the *initialize* method, all Lantz drivers have a *finalize*. Save the python script and run it by:

```
$ python test_fungen.py
```

Note: If you have different versions of python installed, remember to use the one in which you have installed Lantz. You might need to use *python3* instead of *python*.

---

and you will get the following output:

```
FunctionGenerator Serial #12345
```

In the window where *sim-fungen.py* is running you will see the message exchange. You normally don't see this in real instruments. Having a simulated instrument allow us to peek into it and understand what is going on: when we called *inst.idn*, the driver sent message (*?IDN*) to the instrument and it answered back (*FunctionGenerator Serial #12345*). Notice that end of line characters were stripped by the driver.

To find out which other properties and methods are available checkout the documentation. A nice feature of Lantz (thanks to sphinx) is that useful documentation is generated from the driver itself. *idn* is a *Feat* of the driver. Think of a *Feat* as a pimped property. It works just like python properties but it wraps its call with some utilities (more on this later). *idn* is a read-only and as the documentation states it gets the identification information from the device.

Safely releasing resources

As *idn* is read-only, the following code will raise an exception:

```
from lantz.drivers.examples import LantzSignalGeneratorTCP

inst = LantzSignalGeneratorTCP('localhost', 5678)
inst.initialize()
inst.idn = 'A new identification' # <- This will fail as idn is read-only
inst.finalize()
```

The problem is that finalize will never be called possibly leaving resources open. You need to wrap your possible failing code into a try-except-finally structure:

```
from lantz.drivers.examples import LantzSignalGeneratorTCP

inst = LantzSignalGeneratorTCP('localhost', 5678)
inst.initialize()
try:
    inst.idn = 'A new identification' # <- This will fail as idn is read-only
except Exception as e:
    print(e)
finally:
    inst.finalize()
```

All lantz drivers are also context managers and there fore you can write this in a much more compact way:

```
from lantz.drivers.examples import LantzSignalGeneratorTCP

with LantzSignalGeneratorTCP('localhost', 5678) as inst:
    # inst.initialize is called as soon as you enter this block
    inst.idn = 'A new identification' # <- This will fail as idn is read-only
    # inst.finalize is called as soon as you leave this block,
    # even if an error occurs
```

The with statement will create an instance, assign it to *inst* and call *initialize*. The *finalize* will be called independently if there is an exception or not.

Logging

Lantz uses internally the python logging module `logging.Logger`. At any point in your code you can obtain the root Lantz logger:

```
from lantz import LOGGER
```

But additionally, Lantz has some convenience functions to display the log output in a nice format:

```
from lantz.log import log_to_screen, DEBUG, INFO, CRITICAL

from lantz.drivers.examples import LantzSignalGeneratorTCP

# This directs the lantz logger to the console.
log_to_screen(DEBUG)

with LantzSignalGeneratorTCP('localhost', 5678) as inst:
    print(inst.idn)
    print(inst.waveform)
```

Run this script to see the generated log information (it should be colorized in your screen):

```
16:25:03 INFO     Created LantzSignalGeneratorTCP0
16:25:03 DEBUG    Opening port ('localhost', 5678)
16:25:03 INFO     Getting idn
16:25:03 DEBUG    Sending b'?IDN\n'
16:25:03 DEBUG    Received 'FunctionGenerator Serial #12345\n' (len=32)
16:25:03 DEBUG    (raw) Got FunctionGenerator Serial #12345 for idn
16:25:03 INFO     Got FunctionGenerator Serial #12345 for idn
FunctionGenerator Serial #12345
16:25:03 INFO     Getting waveform
16:25:03 DEBUG    Sending b'?WVF\n'
16:25:03 DEBUG    Received '0\n' (len=2)
16:25:03 DEBUG    (raw) Got 0 for waveform
16:25:03 INFO     Got sine for waveform
sine
16:25:03 DEBUG    Closing port ('localhost', 5678)
```

The first line shows the creation of the driver instance. As no name was provided, Lantz assigns one (*LantzSignalGeneratorTCP0*). Line 2 shows that the port was opened (in the implicit call to initialize in the *with* statement). We then request the *idn* (line 3), which is done by sending the command via the TCP port (line 4). 32 bytes are received from the instrument (line 5) which are stripped from the en of line (line 4) and processed (line 6, in this case there is no processing done).

Then the same structure repeats for *waveform*, and important difference is that the driver receives *0* from the instrument and this is translated to the more user friendly *sine*.

Finally, the port is closed (in the implicit call to finalize when leaving the *with* block).

The lines without the time are the result of the print function.

Change *INFO* to *DEBUG* or to *CRITICAL* and run it again to see the different levels of information you can get.

You can change the name of the instrument when you instantiate it:

```
from lantz.log import log_to_screen, DEBUG, INFO, CRITICAL

from lantz.drivers.examples import LantzSignalGeneratorTCP

# This directs the lantz logger to the console.
log_to_screen(DEBUG)

with LantzSignalGeneratorTCP('localhost', 5678) as inst:
    print(inst.idn)
    print(inst.waveform)
```

The cache

As you have seen before, logging provides a look into the Lantz internals. Let's duplicate some code:

```
from lantz.log import log_to_screen, DEBUG

from lantz.drivers.examples import LantzSignalGeneratorTCP

# This directs the lantz logger to the console.
log_to_screen(DEBUG)

with LantzSignalGeneratorTCP('localhost', 5678) as inst:
    print(inst.idn)
    print(inst.idn)
    print(inst.waveform)
    print(inst.waveform)
```

If you see the log output:

```
16:34:40 INFO     Created LantzSignalGeneratorTCP0
16:34:40 DEBUG    Opening port ('localhost', 5678)
16:34:40 INFO     Getting idn
16:34:40 DEBUG    Sending b'?IDN\n'
16:34:40 DEBUG    Received 'FunctionGenerator Serial #12345\n' (len=32)
16:34:40 DEBUG    (raw) Got FunctionGenerator Serial #12345 for idn
16:34:40 INFO     Got FunctionGenerator Serial #12345 for idn
FunctionGenerator Serial #12345
FunctionGenerator Serial #12345
16:34:40 INFO     Getting waveform
16:34:40 DEBUG    Sending b'?WVF\n'
16:34:40 DEBUG    Received '0\n' (len=2)
16:34:40 DEBUG    (raw) Got 0 for waveform
16:34:40 INFO     Got sine for waveform
sine
16:34:40 INFO     Getting waveform
16:34:40 DEBUG    Sending b'?WVF\n'
16:34:40 DEBUG    Received '0\n' (len=2)
16:34:40 DEBUG    (raw) Got 0 for waveform
16:34:40 INFO     Got sine for waveform
```

```
sine
16:34:40 DEBUG    Closing port ('localhost', 5678)
```

idn is only requested once, but waveform twice as you except. The reason is that *idn* is marked *read_once* in the driver as it does not change. The value is cached, preventing unnecessary communication with the instrument.

The cache is specially useful with setters:

```
from lantz.log import log_to_screen, DEBUG

from lantz.drivers.examples import LantzSignalGeneratorTCP

# This directs the lantz logger to the console.
log_to_screen(DEBUG)

with LantzSignalGeneratorTCP('localhost', 5678) as inst:
    inst.waveform = 'sine'
    inst.waveform = 'sine'
```

the log output:

```
16:40:08 INFO     Created LantzSignalGeneratorTCP0
16:40:08 DEBUG    Opening port ('localhost', 5678)
16:40:08 INFO     Setting waveform = sine (current=MISSING, force=False)
16:40:08 DEBUG    (raw) Setting waveform = 0
16:40:08 DEBUG    Sending b'!WVF 0\n'
16:40:08 DEBUG    Received 'OK\n' (len=3)
16:40:08 INFO     waveform was set to sine
16:40:08 INFO     No need to set waveform = sine (current=sine, force=False)
16:40:08 DEBUG    Closing port ('localhost', 5678)
```

Lantz prevents setting the waveform to the same value, a useful feature to speed up communication with instruments in programs build upon decoupled parts.

If you have a good reason to force the change of the value, you can do it with the *update* method:

```
from lantz.log import log_to_screen, DEBUG, INFO, CRITICAL

from lantz.drivers.examples import LantzSignalGeneratorTCP

# This directs the lantz logger to the console.
log_to_screen(DEBUG)

with LantzSignalGeneratorTCP('localhost', 5678) as inst:
    inst.waveform = 'sine'
    inst.update(waveform='sine', force=True)
```

the log output (notice *force=True*):

```
16:41:03 INFO     Created LantzSignalGeneratorTCP0
16:41:03 DEBUG    Opening port ('localhost', 5678)
16:41:03 INFO     Setting waveform = sine (current=MISSING, force=False)
16:41:03 DEBUG    (raw) Setting waveform = 0
16:41:03 DEBUG    Sending b'!WVF 0\n'
16:41:03 DEBUG    Received 'OK\n' (len=3)
16:41:03 INFO     waveform was set to sine
16:41:03 INFO     Setting waveform = sine (current=sine, force=True)
16:41:03 DEBUG    (raw) Setting waveform = 0
16:41:03 DEBUG    Sending b'!WVF 0\n'
16:41:03 DEBUG    Received 'OK\n' (len=3)
```

```
16:41:03 INFO     waveform was set to sine
16:41:03 DEBUG    Closing port ('localhost', 5678)
```

Cache related methods: update, refresh and recall

You have already seen the update method, a method to **set**:

```
inst.waveform = 'sine'
```

is equivalent to:

```
inst.update(waveform='sine')
```

and can also take a dict as an input:

```
inst.update({'waveform': 'sine'})
```

You can also **set** many values at once:

```
inst.update(waveform='sine', amplitude=value)
```

or equivalently:

```
inst.update({'waveform': 'sine'}, 'amplitude': value})
```

but remember that internally these commands will be serialized as not all instruments are capable of dealing with multiple commands.

As you have seen, the update method has a keyword parameter (*force*) that will ignore the current value in the cache.

Lantz also has a method to **get**, named *refresh*:

```
inst.waveform
```

is equivalent to:

```
inst.refresh('waveform')
```

And also work with multiple names:

```
inst.refresh(('frequency', 'amplitude'))
```

or:

```
inst.refresh()
```

to get all values.

In some cases you need the value of some attribute of the instrument that you have not changed since the last time you got/set. The *recall* method returns the value stored in the cache:

```
from lantz.log import log_to_screen, DEBUG

from lantz.drivers.examples import LantzSignalGeneratorTCP

# This directs the lantz logger to the console.
log_to_screen(DEBUG)

with LantzSignalGeneratorTCP('localhost', 5678) as inst:
    print(inst.waveform)
    print(inst.recall('waveform'))
```

You can use the the driver that you have created in you projects. Learn more in the next part of the tutorial: *Using Feats*.

3.3.3 Using Feats

Let's query all parameters and print their state in a nice format:

```
from lantz.drivers.examples import LantzSignalGeneratorTCP

with LantzSignalGeneratorTCP('localhost', 5678) as inst:
    print('idn: {}'.format(inst.idn))
    print('frequency: {}'.format(inst.frequency))
    print('amplitude: {}'.format(inst.amplitude))
    print('offset: {}'.format(inst.offset))
    print('output_enabled: {}'.format(inst.output_enabled))
    print('waveform: {}'.format(inst.waveform))
    for channel in range(1, 9):
        print('dout[{}]: {}'.format(channel, inst.dout[channel]))
    for channel in range(1, 9):
        print('din[{}]: {}'.format(channel, inst.din[channel]))
```

If you run the program you will get something like:

```
idn: FunctionGenerator Serial #12345
frequency: 1000.0 hertz
amplitude: 0.0 volt
offset: 0.0 volt
output_enabled: False
waveform: sine
dout[1]: False
dout[2]: False
dout[3]: False
dout[4]: False
dout[5]: False
dout[6]: False
dout[7]: False
dout[8]: False
din[1]: False
din[2]: False
din[3]: False
din[4]: False
din[5]: False
din[6]: False
din[7]: False
din[8]: False
```

Valid values

You can set property like *output_enabled*:

```
from lantz.drivers.examples import LantzSignalGeneratorTCP

with LantzSignalGeneratorTCP('localhost', 5678) as inst:
    print('output_enabled: {}'.format(inst.output_enabled))
    inst.output_enabled = True
    print('output_enabled: {}'.format(inst.output_enabled))
```

If you check the documentation for lantz.drivers.examples.LantzSignalGeneratorTCP), *output_enabled* accepts only *True* or *False*. If your provide a different value:

```
from lantz.drivers.examples import LantzSignalGeneratorTCP

with LantzSignalGeneratorTCP('localhost', 5678) as inst:
    inst.output_enabled = 'Yes'
```

you will get an error message:

```
Traceback (most recent call last):
  File "using7.py", line 5, in <module>
    inst.output_enabled = 'Yes'
...
ValueError: 'Yes' not in (False, True)
```

Units

Feats corresponding to physical quantities (magnitude and units), are declared with a default unit. If try to set a number to them:

```
from lantz.drivers.examples import LantzSignalGeneratorTCP

with LantzSignalGeneratorTCP('localhost', 5678) as inst:
    inst.amplitude = 1
```

Lantz will issue a warning:

```
DimensionalityWarning: Assuming units `volt` for 1
```

Lantz uses the Pint package to declare units:

```
from lantz.drivers.examples import LantzSignalGeneratorTCP
from lantz import Q_

with LantzSignalGeneratorTCP('localhost', 5678) as inst:
    inst.amplitude = Q_(1, 'Volts')
    print('amplitude: {}'.format(inst.amplitude))
```

the output is:

```
amplitude: 1.0 volt
```

The nice thing is that this will work even if the instruments and you program opeate in different units. The conversion is done internally, minimizing errors and allowing better interoperability:

```
from lantz.drivers.examples import LantzSignalGeneratorTCP
from lantz import Q_

with LantzSignalGeneratorTCP('localhost', 5678) as inst:
    inst.amplitude = Q_(.1, 'decivolt')
    print('amplitude: {}'.format(inst.amplitude))
```

the output is:

```
amplitude: 0.1 volt
```

Numerical Feats can also define the valid limits, for amplitude is 0 - 10 Volts. If you provide a value out of range:

```
inst.amplitude = Q_(20, 'volt')
```

you get:

```
Traceback (most recent call last):
  File "using10.py", line 6, in <module>
    inst.amplitude = Q_(20, 'volt')
...
ValueError: 20 not in range (0, 10)
```

While Lantz aims to provide drivers for most common instruments, sometimes you will need to build your own drivers. Learn how in the next part of the tutorial: *Building your own drivers*.

3.3.4 Building your own drivers

In this part of the tutorial, we are going to build the driver of an hypothetical signal generator. Following a tutorial about building a driver to communicate with an instrument that you do not have is not much fun. That's why we have created a virtual version of this instrument. From the command line, run the following command:

```
$ lantz-sim fungen tcp
```

This will start an application that listens for incoming TCP packages on port 5678 from *localhost*.

Note: If you have done the previous tutorial, you will build from scratch the same driver that is included in Lantz.

The instrument

The signal generator has the following characteristics:

- 1 Analog output
  - Frequency range: 1 Hz to 100 KHz
  - Amplitude (0-Peak): 0 V to 10 V
  - Offset: -5V to 5V
  - Waveforms: sine, square, triangular, ramp
- 8 Digital outputs
- 8 Digital inputs

Your program will communicate with the instrument communicates exchanging messages via TCP protocol over ethernet. Messages are encoding in ASCII and line termination is `LF` (Line feed, 'n', 0x0A, 10 in decimal) for both sending and receiving.

The following commands are defined:

Command	Description	Example command	Example response
?IDN	Get identification	?IDN	LSG Serial #1234
?FRE	Get frequency [Hz]	?FRE	233.34
?AMP	Get amplitude [V]	?AMP	8.3
?OFF	Get offset [V]	?OFF	1.7
?OUT	Get output enabled state	?OUT	1
?WVF *W*	Get waveform	?WVF	2
?DOU *D*	Get digital output state	?DOU 4	0
?DIN *D*	Get digital input state	?DIN 19	ERROR
!FRE *F*	Set frequency [Hz]	!FRE 20.80	OK
!AMP *F*	Set amplitude [V]	!AMP 11.5	ERROR
!OFF *F*	Set offset [V]	!OFF -1.2	OK
!WVF *W*	Set waveform	!WVF 3	OK
!OUT *B*	Set output enabled state	!OUT 0	OK
!DOU *D B*	Set digital output state	!DOU 4 1	OK
!CAL	Calibrate system	!CAL	OK

As shown in the table, commands used to get the state of the instrument start with **?** and commands used to set the state start with **!**. In the **Command** column:

- *D* is used to indicate the digital input or output channel being addressed (1-8)
- *F* is the value of a float parameter. The actual valid range for each parameter depends on the command itself.
- *W* is used to indicate the desired waveform (0: sine, 1:square, 2:triangular, 3: ramp)
- *B* is the state of the digital input or output channel (0 is off/low, 1 is on/high), or the state of the analog ourput (0 off/disabled, 1 on/enabled)

The response to successful **GET** commands is the requested value. The response to successful **SET** commands is the string OK. If the command is invalid or an occurs in the instrument, the instrument will respond with the string ERROR. For example, the command `?DIS 19` is invalid because the parameter *B* should be in [1, 8].

A basic driver

Having look at the instrument, we will now create the driver. Open the project folder that you created in the previous tutorial (*I*
it is yours) and change it to look like this:

```
from lantz import Feat
from lantz.network import TCPDriver

class LantzSignalGeneratorTCP(TCPDriver):
    """Lantz Signal Generator.
    """

    ENCODING = 'ascii'

    RECV_TERMINATION = '\n'
    SEND_TERMINATION = '\n'

    @Feat()
    def idn(self):
        return self.query('?IDN')


if __name__ == '__main__':
    with LantzSignalGeneratorTCP('localhost', 5678) as inst:
        print('The identification of this instrument is : ' + inst.idn)
```

The code is straight forward. We first import TCPDriver from lantz.network (the Lantz module for network related functions). TCPDriver is a base class (derived from Driver) that implements methods to communicate via TCP protocol. Our driver will derive from this.

We also import Feat from lantz. Feat is the Lantz pimped property and you use Feat just like you use *property*. By convention Feats are named using nouns or adjectives. Inside the method (in this case is a getter) goes the code to communicate with the instrument. In this case we use *query*, a function present in all based classes for message drivers (TCPDriver, SerialDriver, etc). *query* sends a message to the instrument, waits for a response and returns it. The argument is the command to be sent to the instrument. Lantz takes care of formatting (encoding, endings) and transmitting the command appropriately. That's why we define ENCODING, RECV_TERMINATION, SEND_TERMINATION at the beginning of the class.

Finally, inside the *__name__ == '__main__'* we instantiate the SignalGenerator specifying host and port (these are arguments of the TCPDriver constructor, more on this later) and we print the identification.

If you have the simulator running, you can test your new driver. From the command line, cd into the project directory and then run the following command:

```
$ python mydriver.py
```

Note: If you have different versions of python installed, remember to use the one in which you have installed Lantz. You might need to use *python3* instead of *python*.

You should see *LSG Serial #1234*.

Let's see what's its going on under the hood by logging to screen in debug mode:

```
from lantz.log import log_to_screen, DEBUG  # <-- This is new

from lantz import Feat
from lantz.network import TCPDriver

class LantzSignalGeneratorTCP(TCPDriver):
    """Lantz Signal Generator.
    """

    ENCODING = 'ascii'

    RECV_TERMINATION = '\n'
    SEND_TERMINATION = '\n'

    @Feat()
    def idn(self):
        """Identification.
        """
        return self.query('?IDN')


if __name__ == '__main__':
    log_to_screen(DEBUG)
    with LantzSignalGeneratorTCP('localhost', 5678) as inst:
        print('The identification of this instrument is : ' + inst.idn)
```

You can adjust the level of information provided by changing the LOGGING_LEVEL. You can also display the logging in another window to avoid cluttering but this comes later.

Let's allow our driver to control the instruments amplitude:

```
from lantz import Feat
from lantz.network import TCPDriver

class LantzSignalGeneratorTCP(TCPDriver):
    """Lantz Signal Generator.
    """

    ENCODING = 'ascii'

    RECV_TERMINATION = '\n'
    SEND_TERMINATION = '\n'

    @Feat()
    def idn(self):
        """Identification.
        """
        return self.query('?IDN')


    @Feat()
    def amplitude(self):
        """Amplitude (0 to peak) in volts.
        """
        return float(self.query('?AMP'))

    @amplitude.setter
    def amplitude(self, value):
        self.query('!AMP {:.1f}'.format(value))


if __name__ == '__main__':
    from time import sleep
    from lantz.log import log_to_screen, DEBUG

    log_to_screen(DEBUG)
    with LantzSignalGeneratorTCP('localhost', 5678) as inst:
        print('The identification of this instrument is : ' + inst.idn)
        print('Setting amplitude to 3')
        inst.amplitude = 3
        sleep(2)
        inst.amplitude = 5
        print('Current amplitude: {}'.format(inst.amplitude))
```

We have defined another Feat, now with a getter and a setter. The getter sends *?AMP* and waits for the answer which is converted to float and returned to the caller. The setter send *!AMP* concatenated with the float formatted to string with two decimals. Run the script. Check also the window running *sim-fungen.py*. You should see the amplitude changing!.

In the current version of this driver, if we try to set the amplitude to 20 V the command will fill in the instrument but the driver will not know. Lets add some error checking:

```
# import ...

class LantzSignalGeneratorTCP(TCPDriver):

    # Code from previous example
    # ...

    @amplitude.setter
    def amplitude(self, value):
```

```
        if self.query('!AMP {:.2f}'.format(value)) != "OK":
            raise Exception
```

Is that simple. We just check the response. If different from *OK* we raise an Exception. Change the script to set the amplitude to 20 and run it one more time. You should something like this in the log:

```
Exception: While setting amplitude to 20.
```

We do not know why the command has failed but we know which command has failed.

Because all commands should be checked for *ERROR*, we will override query to do it. Reset amplitude to the original and add the following, add the following import to the top of the file, and redefine the query function to the class:

```
# import ...
from lantz.errors import InstrumentError

class LantzSignalGeneratorTCP(TCPDriver):

    # Code from previous example
    # ...

    @amplitude.setter
    def amplitude(self, value):
        self.query('!AMP {:.1f}'.format(value))

    def query(self, command, *, send_args=(None, None), recv_args=(None, None)):
        answer = super().query(command, send_args=send_args, recv_args=recv_args)
        if answer == 'ERROR':
            raise InstrumentError
        return answer
```

The *query* function mediates all queries to the instrument. In our redefined version, we call the original first (*super().query(...)*) and then we check for errors. In this way we have added error checking for all queries!.

Putting units to work

Hoping that the Mars Orbiter story convinced you that using units is worth it, let's modify the driver to use them:

```
from lantz import Feat
from lantz.network import TCPDriver
from lantz.errors import InstrumentError

class LantzSignalGeneratorTCP(TCPDriver):
    """Lantz Signal Generator.
    """

    ENCODING = 'ascii'

    RECV_TERMINATION = '\n'
    SEND_TERMINATION = '\n'

    def query(self, command, *, send_args=(None, None), recv_args=(None, None)):
        answer = super().query(command, send_args=send_args, recv_args=recv_args)
        if answer == 'ERROR':
            raise InstrumentError
        return answer

    @Feat()
```

```
    def idn(self):
        return self.query('?IDN')


    @Feat(units='V')
    def amplitude(self):
        """Amplitude (0 to peak)
        """
        return float(self.query('?AMP'))

    @amplitude.setter
    def amplitude(self, value):
        self.query('!AMP {:.1f}'.format(value))


if __name__ == '__main__':
    from time import sleep
    from lantz import Q_
    from lantz.log import log_to_screen, DEBUG

    volt = Q_(1, 'V')
    milivolt = Q_(1, 'mV')

    log_to_screen(DEBUG)
    with LantzSignalGeneratorTCP('localhost', 5678) as inst:
        print('The identification of this instrument is : ' + inst.idn)
        print('Setting amplitude to 3')
        inst.amplitude = 3 * volt
        sleep(2)
        inst.amplitude = 1000 * milivolt
        print('Current amplitude: {}'.format(inst.amplitude))
```

We have just added in the Feat definition that the units is Volts. Lantz uses the Pint package to manage units. We now import *Q_* which is a shortcut for *Pint.Quantity* and we declare the volt and the milivolt. We now set the amplitude to 3 Volts and 1000 milivolts.

Run the script and notice how Lantz will do the conversion for you. This allows to use the output of one instrument as the output of another without handling the unit conversion. Additionally, it allows you to replace this signal generator by another that might require the amplitude in different units without changing your code.

Limits

When the communication round-trip to the instrument is too long, you might want to catch some of the errors before hand. You can use *limits* to check for valid range of the parameter. Limits syntax is:

```
limits([start,] stop[, step])

limits(10)          # means from 0 to 10 (the 10 is valid)
limits(2, 10)       # means from 2 to 10 (the 10 is valid)
limits(2, 10, 2)    # means from 2 to 10, with a step of 2 (the 10 is valid)
```

If you provide a value outside the valid range, Lantz will raise a ValueError. If the steps parameter is set but you provide a value not compatible with it, it will be silently rounded. Let's put this to work for amplitude, frequency and offset:

```
from lantz import Feat
from lantz.network import TCPDriver
```

```
from lantz.errors import InstrumentError

class LantzSignalGeneratorTCP(TCPDriver):
    """Lantz Signal Generator
    """

    ENCODING = 'ascii'

    RECV_TERMINATION = '\n'
    SEND_TERMINATION = '\n'

    def query(self, command, *, send_args=(None, None), recv_args=(None, None)):
        answer = super().query(command, send_args=send_args, recv_args=recv_args)
        if answer == 'ERROR':
            raise InstrumentError
        return answer

    @Feat()
    def idn(self):
        return self.query('?IDN')

    @Feat(units='V', limits=(10,)) # This means 0 to 10
    def amplitude(self):
        """Amplitude.
        """
        return float(self.query('?AMP'))

    @amplitude.setter
    def amplitude(self, value):
        self.query('!AMP {:.1f}'.format(value))

    @Feat(units='V', limits=(-5, 5, .01)) # This means -5 to 5 with step 0.01
    def offset(self):
        """Offset
        """
        return float(self.query('?OFF'))

    @offset.setter
    def offset(self, value):
        self.query('!OFF {:.1f}'.format(value))

    @Feat(units='Hz', limits=(1, 1e+5)) # This means 1 to 1e+5
    def frequency(self):
        """Frequency
        """
        return float(self.query('?FRE'))

    @frequency.setter
    def frequency(self, value):
        self.query('!FRE {:.2f}'.format(value))
```

If you try to set a value outside the valid range, a ValueErorr will be raised and the command will never be sent to the instrument. Give it a try:

```
inst.amplitude = 20
```

Automatic rounding:

```
inst.offset = 0.012 # rounded to 0.01
```

Mapping values

We will define offset and frequency like we did with amplitude, and we will also define output enabled and waveform:

```
from lantz import Feat, DictFeat
from lantz.network import TCPDriver
from lantz.errors import InstrumentError

class LantzSignalGeneratorTCP(TCPDriver):
    """Lantz Signal Generator
    """

    ENCODING = 'ascii'

    RECV_TERMINATION = '\n'
    SEND_TERMINATION = '\n'

    def query(self, command, *, send_args=(None, None), recv_args=(None, None)):
        answer = super().query(command, send_args=send_args, recv_args=recv_args)
        if answer == 'ERROR':
            raise InstrumentError
        return answer

    @Feat()
    def idn(self):
        return self.query('?IDN')

    @Feat(units='V', limits=(10,))
    def amplitude(self):
        """Amplitude.
        """
        return float(self.query('?AMP'))

    @amplitude.setter
    def amplitude(self, value):
        self.query('!AMP {:.1f}'.format(value))

    @Feat(units='V', limits=(-5, 5, .01))
    def offset(self):
        """Offset.
        """
        return float(self.query('?OFF'))

    @offset.setter
    def offset(self, value):
        self.query('!OFF {:.1f}'.format(value))

    @Feat(units='Hz', limits=(1, 1e+5))
    def frequency(self):
        """Frequency.
        """
        return float(self.query('?FRE'))

    @frequency.setter
    def frequency(self, value):
```

```
        self.query('!FRE {:.2f}'.format(value))

    @Feat(values={True: 1, False: 0})
    def output_enabled(self):
        """Analog output enabled.
        """
        return int(self.query('?OUT'))

    @output_enabled.setter
    def output_enabled(self, value):
        self.query('!OUT {}'.format(value))

    @Feat(values={'sine': 0, 'square': 1, 'triangular': 2, 'ramp': 3})
    def waveform(self):
        return int(self.query('?WVF'))

    @waveform
    def waveform(self, value):
        self.query('!WVF {}'.format(value))

if __name__ == '__main__':
    from time import sleep
    from lantz import Q_
    from lantz.log import log_to_screen, DEBUG

    volt = Q_(1, 'V')
    milivolt = Q_(1, 'mV')
    Hz = Q_(1, 'Hz')

    log_to_screen(DEBUG)
    with LantzSignalGeneratorTCP('localhost', 5678) as inst:
        print('The identification of this instrument is : ' + inst.idn)
        print('Setting amplitude to 3')
        inst.amplitude = 3 * volt
        inst.offset = 200 * milivolt
        inst.frequency = 20 * Hz
        inst.output_enabled = True
        inst.waveform = 'sine'
```

We have provided *output_enabled* a mapping table through the *values* argument. This has two functions:

- Restricts the input to True or False.
- For the setter converts True and False to 1 and 0; and vice versa for the getter.

This means that we can write the body of the getter/setter expecting a instrument compatible value (1 or 0) but the user actually sees a much more friendly interface (True or False). The same happens with *waveform*. Instead of asking the user to memorize which number corresponds to 'sine' or implement his own mapping, we provide this within the feat.

Properties with items: DictFeat

It is quite common that scientific equipment has many of certain features (such as axes, channels, etc). For example, this signal generator has 8 digital outputs. A simple solution would be to access them as feats named dout1, dout2 and so on. But this is not elegant (consider a DAQ with 32 digital inputs) and makes coding to programatically access to channel N very annoying. To solve this Lantz provides a dictionary like feature named `DictFeat`. Let's see this in action:

```
# import ...

class LantzSignalGeneratorTCP(TCPDriver):

    # Code from previous example
    # ...

    @DictFeat(values={True: 1, False: 0})
    def dout(self, key):
        """Digital output state.
        """
        return int(self.query('?DOU {}'.format(key)))

    @dout.setter
    def dout(self, key, value):
        self.query('!DOU {} {}'.format(key, value))
```

In the driver definition, very little has changed. `DictFeat` acts like the standard Feat decorator but operates on a method that contains one extra parameter for the get and the set in the second position.

You will use this in the following way:

```
inst.dout[4] = True
```

By default, any key (in this case, channel) is valid and Lantz leaves to the underlying instrument to reject invalid ones. In some cases, for example when the instrument does not deal properly with unexpected parameters, you might want to restrict them using the optional parameter *keys*

```
# import ...

class LantzSignalGeneratorTCP(TCPDriver):

    # Code from previous example
    # ...

    @DictFeat(values={True: 1, False: 0}, keys=list(range(1,9)))
    def dout(self, key):
        """Digital output state.
        """
        return int(self.query('?DOU {}'.format(key)))

    @dout.setter
    def dout(self, key, value):
        self.query('!DOU {} {}'.format(key, value))
```

Remember that range(1, 9) excludes 9. In this way, Lantz will Raise an exception without talking to the instrument when the following code:

```
>>> inst.dout[10] = True
Traceback:
    ...
KeyError: 10 is not valid key for dout [1, 2, 3, 4, 5, 6, 7, 8]
```

We will create now a read-read only DictFeat for the digital input:

```
# import ...

class LantzSignalGeneratorTCP(TCPDriver):
```

```
    # Code from previous example
    # ...

    @DictFeat(values={True: 1, False: 0}, keys=list(range(1,9)))
    def din(self, key):
        """Digital input state.
        """
        return int(self.query('?DIN {}'.format(key)))
```

Drivers methods: Action

Bound methods that will trigger interaction with the instrument are decorated with `Action`:

```
from lantz import Feat, DictFeat, Action
```

and within the class we will add:

```
@Action()
def calibrate(self):
    self.query('!CAL')
```

You can use the the driver that you have created in you projects. Learn how in the next part of the tutorial: *A simple command line app*.

3.3.5 A simple command line app

In this part of the tutorial you will build a simple command line application to do a frequency scan.

Start the simulated instrument running the following command:

```
$ lantz-sim fungen tcp
```

Open the folder in which you have created the driver (*Building your own drivers*) and create a python file named *scanfrequency.py*:

```
import time

from lantz import Q_

from mydriver import LantzSignalGeneratorTCP

Hz = Q_(1, 'Hz')
start = 1 * Hz
stop = 10 * Hz
step = 1 * Hz
wait = .5

with LantzSignalGeneratorTCP('localhost', 5678) as inst:
    print(inst.idn)

    current = start

    # This loop scans the frequency
    while current < stop:
        inst.frequency = current
        print('Changed to {}'.format(current))
```

```
        time.sleep(wait)
        current += step
```

First we the *time* module, the quantities class and the driver that you created in the previous tutorial. We could have used the driver included in Lantz, but we will work as if the driver was not in Lantz and you have built it yourself for your project. We create an instance of it using a context manager (the *with* statement) to make sure that all resources will be properly closed even if an error occurs. Finally, we just step through all the frequencies, changing the instrument and waiting at each step.

Run it using:

```
$ python scanfrequency.py
```

Using command line arguments

In our first implementation the scan range and the waiting time were fixed. We will now add mandatory command line arguments to set the start and stop frequency and optionally the step size and the waiting time. To do this, we will import the *argparse* module and create a parser object:

```
import time
import argparse

from lantz import Q_

from mydriver import LantzSignalGeneratorTCP

parser = argparse.ArgumentParser()
parser.add_argument('start', type=float,
                    help='Start frequency [Hz]')
parser.add_argument('stop', type=float,
                    help='Stop frequency [Hz]')
parser.add_argument('step', type=float,
                    help='Step frequency [Hz]')
parser.add_argument('wait', type=float,
                    help='Waiting time at each step [s]')

args = parser.parse_args()

Hz = Q_(1, 'Hz')
start = args.start * Hz
stop = args.stop * Hz
step = args.step * Hz
wait = args.wait

with LantzSignalGeneratorTCP('localhost', 5678) as inst:
    print(inst.idn)

    current = start
    while current < stop:
        inst.frequency = current
        print('Changed to {}'.format(current))
        time.sleep(wait)
        current += step
```

A nice thing about Python argparse package is that you get the help:

```
$ python scanfrequency.py
```

or in more detail:

```
python scanfrequency.py -h
```

Try it again specifying the start, stop, step and waiting time:

```
$ python scanfrequency.py 2 8 2 .1
```

Refactoring for reusability

Finally we will add a couple of lines to allow the user to define the host and port number of the TCP function generator. We will also refactor the code to extract the function that perform the actual frequency scan apart:

```
import time

def scan_frequency(inst, start, stop, step, wait):
    """Scan frequency in an instrument.

    :param start: Start frequency.
    :type start: Quantity
    :param stop: Stop frequency.
    :type stop: Quantity
    :param step: Step frequency.
    :type step: Quantity
    :param wait: Waiting time.
    :type wait: Quantity

    """
    in_secs = wait.to('seconds').magnitude
    current = start
    while current < stop:
        inst.frequency = current
        time.sleep(in_secs)
        current += step


if __name__ == '__main__':
    import argparse

    from lantz import Q_

    from mydriver import LantzSignalGeneratorTCP

    parser = argparse.ArgumentParser()

    # Configure
    parser.add_argument('-H', '--host', type=str, default='localhost',
                        help='TCP hostname')
    parser.add_argument('-p', '--port', type=int, default=5678,
                        help='TCP port')

    parser.add_argument('start', type=float,
                        help='Start frequency [Hz]')
    parser.add_argument('stop', type=float,
                        help='Stop frequency [Hz]')
```

```
    parser.add_argument('step', type=float,
                        help='Step frequency [Hz]')
    parser.add_argument('wait', type=float,
                        help='Waiting time at each step [s]')

    args = parser.parse_args()

    Hz = Q_(1, 'Hz')
    sec = Q_(1, 'sec')

    def print_change(new, old):
        print('Changed from {} to {}'.format(old, new))

    with LantzSignalGeneratorTCP(args.host, args.port) as inst:
        print(inst.idn)

        inst.frequency_changed.connect(print_change)

        scan_frequency(inst, args.start * Hz, args.stop * Hz,
                       args.step * Hz, args.wait * sec)
```

The first change you will notice is that we have now used a Quantity for the time. It might be meaningless as the script ask for the waiting time in seconds and the function used to wait (*time.sleep*) expects the time in seconds. But using a Quantity allows the caller of the function how the waiting is implemented.

Also notice that we have removed the print statement from inside the function to be able to reuse it in other applications. For example, we might want to use it in a silent command line application or in a GUI application. To know that the frequency has changed we have connected a reporting function (*print_change*) to a signal (*frequency_changed*). Lantz will call the function every time that the frequency changes. Every Feat has an associated signal that can be accessed by appending *_changed* to the name.

If you have installed PyQt4 (or PySide) you can use Lantz helpers to build a GUI app. Learn how in the next part of the tutorial: *A simple GUI app*.

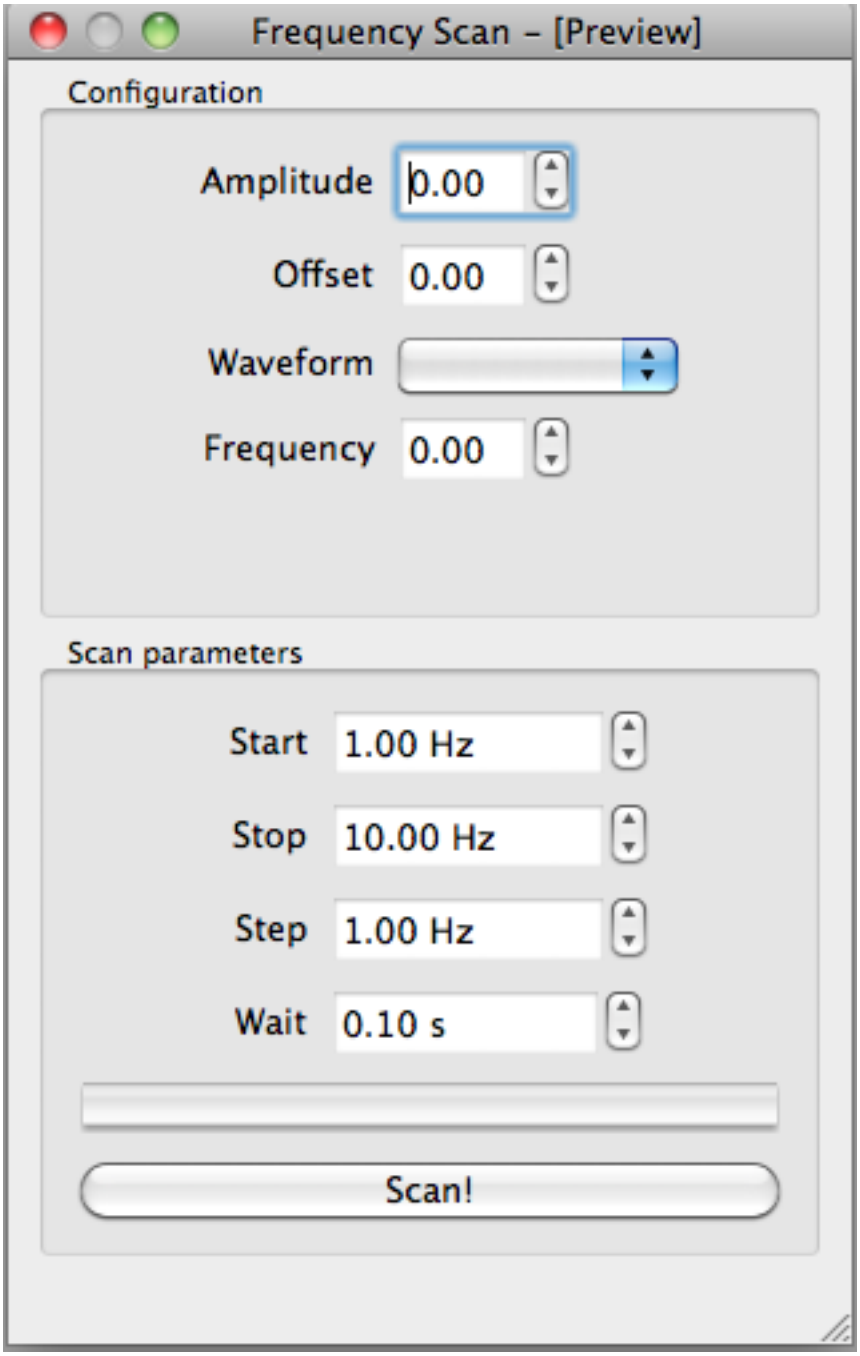
3.3.6 A simple GUI app

In this part of the tutorial you will build an application to do a frequency scan like in the previous tutorial (*A simple command line app*) but with a graphical frontend.

Start the simulated instrument running the following command:

```
$ lantz-sim fungen tcp
```

Using Qt Designer, create a window like this:



and save it as *scanfrequency.ui* in the folder in which you have created the driver (*Building your own drivers*). For the example, we have labeled each control as corresponding label in lower caps (amplitude, offset, waveform, start, stop, step, wait). The button is named *scan*. You can also download `the ui file` if you prefer.

Notice that the *amplitude* and *offset* don't show units and that the *waveform* combobox is not populated. These widgets will be connected to the corresponding Feats of the drivers and Lantz will take care of setting the right values, items, etc.

Create a python file named *scanfrequency-gui.py* with the following content:

```
import sys
```

```
# Import lantz.ui register an import hook that will replace calls to Qt by PyQt4 or PySide ...
import lantz.ui

# and here we just use Qt and will work with both bindings!
from Qt.QtGui import QApplication, QWidget
from Qt.uic import loadUi

app = QApplication(sys.argv)

# Load the UI from the QtDesigner file. You can also use pyuic4 to generate a class.
main = loadUi('scanfrequency.ui')

# Start the app
main.show()
exit(app.exec_())
```

Run this stub should display the window:

```
$ python scanfrequency-gui.py
```

Note: In Windows, you can use *pythonw* instead of *python* to suppress the terminal window.

We will now add the code to do the actual frequency scan. We will reuse the function from the last tutorial:

```
import sys

# Import lantz.ui register an import hook that will replace calls to Qt by PyQt4 or PySide ...
import lantz.ui
from lantz import Q_

# and here we just use Qt and will work with both bindings!
from Qt.QtGui import QApplication, QWidget
from Qt.uic import loadUi

# These imports are from your own project
from mydriver import LantzSignalGeneratorTCP
from scanfrequency import scan_frequency

app = QApplication(sys.argv)

# Load the UI from the QtDesigner file. You can also use pyuic4 to generate a class.
main = loadUi('scanfrequency.ui')

Hz = Q_(1, 'Hz')
sec = Q_(1, 'second')

with LantzSignalGeneratorTCP('localhost', 5678) as inst:

    # Obtain a reference to the widgets controlling the scan parameters
    start = main.findChild((QWidget, ), 'start')
    stop = main.findChild((QWidget, ), 'stop')
    step = main.findChild((QWidget, ), 'step')
    wait = main.findChild((QWidget, ), 'wait')
    scan = main.findChild((QWidget, ), 'scan')

    # Define a function to read the values from the widget and call scan_frequency
    def scan_clicked():
```

```
        scan_frequency(inst, start.value() * Hz, stop.value() * Hz,
                       step.value() * Hz, wait.value() * sec)

    # Connect the clicked signal of the scan button to the function
    scan.clicked.connect(scan_clicked)

    # Scan the app
    main.show()
    exit(app.exec_())
```

When the button is clicked, Qt will emit a signal which is connected to the function we have defined the application should scan the frequency. You will not see anything happening in the Window, but if you look in the simulator console you will see the frequency changing.

Connecting widgets to Feats

To allow the user to change the amplitude, offset, shape and frequency, we will connect the configuration widgets:

```
import sys

# Import lantz.ui register an import hook that will replace calls to Qt by PyQt4 or PySide ...
import lantz.ui
from lantz import Q_

# and here we just use Qt and will work with both bindings!
from Qt.QtGui import QApplication, QWidget
from Qt.uic import loadUi

# Import from lantz a function to connect drivers to UI <--- NEW
from lantz.ui.qtwidgets import connect_driver

# These imports are from your own project
from mydriver import LantzSignalGeneratorTCP
from scanfrequency import scan_frequency

app = QApplication(sys.argv)

# Load the UI from the QtDesigner file. You can also use pyuic4 to generate a class.
main = loadUi('scanfrequency.ui')

Hz = Q_(1, 'Hz')
sec = Q_(1, 'second')

with LantzSignalGeneratorTCP('localhost', 5678) as inst:

    # Obtain a reference to the widgets controlling the scan parameters
    start = main.findChild((QWidget, ), 'start')
    stop = main.findChild((QWidget, ), 'stop')
    step = main.findChild((QWidget, ), 'step')
    wait = main.findChild((QWidget, ), 'wait')
    scan = main.findChild((QWidget, ), 'scan')

    # <--------- This is new --------->
    connect_driver(main, inst)

    progress = main.findChild((QWidget, ), 'progress')
```

```
    def update_progress_bar(new, old):
        fraction = (new.magnitude - start.value()) / (stop.value() - start.value())
        progress.setValue(fraction * 100)

    inst.frequency_changed.connect(update_progress_bar)


    # Define a function to read the values from the widget and call scan_frequency
    def scan_clicked():
        scan_frequency(inst, start.value() * Hz, stop.value() * Hz,
                       step.value() * Hz, wait.value() * sec)

    # Connect the clicked signal of the scan button to the function
    scan.clicked.connect(scan_clicked)

    # Scan the app
    main.show()
    exit(app.exec_())
```

The function *connect_driver* matches by name Widgets to Feats and then connects them. Under the hood, for each match it:

1.- Wraps the widget to make it Lantz compatible.

2.- If applicable, configures minimum, maximum, steps and units.

3.- Add a handler such as when the widget value is changed, the Feat is updated.

4.- Add a handler such as when the Feat value is changed, the widget is updated.

You can learn more and some alternatives in *Connecting a custom UI to a driver*.

To update the progress bar, we connected the *frequency_changed* signal to a function that updates the progress bar.

Run this example and test how you can change the amplitude, offset and waveform:

```
$ python scanfrequency-gui.py
```

However, you will see that the frequency and the progress bar are not updated during the scan.

Using a background thread

The drawback of the previous (simple) approach is that the scan is executed in the same thread as the GUI, effectively locking the main window and making the application unresponsive. Qt Multithreading programming is out of the scope of this tutorial (checkout **'Threads in Qt'_** for more info), but we will provide some examples how to do it:

```
import sys

# Import lantz.ui register an import hook that will replace calls to Qt by PyQt4 or PySide ...
import lantz.ui
from lantz import Q_

# Import from lantz a function to connect drivers to UI
from lantz.ui.qtwidgets import connect_driver

# and here we just use Qt and will work with both bindings!
from Qt.QtGui import QApplication, QWidget
from Qt.uic import loadUi

# We import
```

```
from Qt.QtCore import QThread, QObject

# These imports are from your own project
from mydriver import LantzSignalGeneratorTCP
from scanfrequency import scan_frequency

app = QApplication(sys.argv)

# Load the UI from the QtDesigner file. You can also use pyuic4 to generate a class.
main = loadUi('scanfrequency.ui')

Hz = Q_(1, 'Hz')
sec = Q_(1, 'second')

with LantzSignalGeneratorTCP('localhost', 5678) as inst:

    # Connect the main panel widgets to the instruments Feats,
    # matching by name
    connect_driver(main, inst)

    # Obtain a reference to the widgets controlling the scan parameters
    start = main.findChild((QWidget, ), 'start')
    stop = main.findChild((QWidget, ), 'stop')
    step = main.findChild((QWidget, ), 'step')
    wait = main.findChild((QWidget, ), 'wait')
    scan = main.findChild((QWidget, ), 'scan')
    progress = main.findChild((QWidget, ), 'progress')

    def update_progress_bar(new, old):
        fraction = (new.magnitude - start.value()) / (stop.value() - start.value())
        progress.setValue(fraction * 100)

    inst.frequency_changed.connect(update_progress_bar)

    # <--------- New code--------->
    # Define a function to read the values from the widget and call scan_frequency
    class Scanner(QObject):

        def scan(self):
            # Call the scan frequency
            scan_frequency(inst, start.value() * Hz, stop.value() * Hz,
                           step.value() * Hz, wait.value() * sec)
            # When it finishes, set the progress to 100%
            progress.setValue(100)

    thread = QThread()
    scanner = Scanner()
    scanner.moveToThread(thread)
    thread.start()

    # Connect the clicked signal of the scan button to the function
    scan.clicked.connect(scanner.scan)

    app.aboutToQuit.connect(thread.quit)
    # <--------- End of new code --------->

    main.show()
    exit(app.exec_())
```

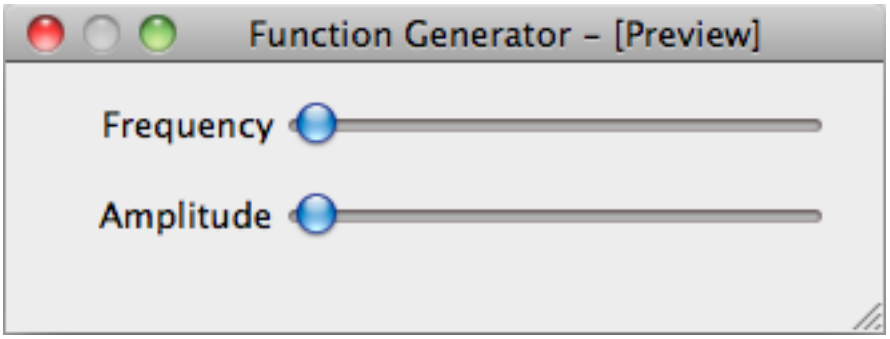
In Qt, when a signal is connected to a slot (a function of a QObject), the execution occurs in the Thread of the receiver (not the emitter). That is why we moved the QObject to the new thread.

Note: On a production app it would be good to add a lock to prevent the application from exiting or calling the scanner while a scanning is running.

3.4 Guides

3.4.1 Connecting a custom UI to a driver

While the test widget is very convenient is not good enough for visually attractive applications. You can design you own custom user interface using Qt Designer and then connect it to your driver in a very simple way. Consider the following interace for our custom signal generator.



You can set the frequency and amplitude using sliders. The sliders are named *frequency* and *amplitude*.

The long way

You can connect each relevant driver Feat to the corresponding widget:

```
import sys

# Import lantz.ui register an import hook that will replace calls to Qt by PyQt4 or PySide ...
import lantz.ui

# and here we just use Qt and will work with both bindings!
from Qt.QtGui import QApplication, QWidget
from Qt.uic import loadUi

# From lantz we import the driver ...
from lantz.drivers.examples.fungen import LantzSignalGeneratorTCP

# and a function named connect_feat that does the work.
from lantz.ui.qtwidgets import connect_feat

app = QApplication(sys.argv)

# We load the UI from the QtDesigner file. You can also use pyuic4 to generate a class.
main = loadUi('connect_test.ui')

# We get a reference to each of the widgets.
```

```
frequency_widget = main.findChild((QWidget, ), 'frequency')
amplitude_widget = main.findChild((QWidget, ), 'amplitude')

with LantzSignalGeneratorTCP('localhost', 5678) as inst:

    # We connect each widget to each feature
    # The syntax arguments are widget, target (driver), Feat name
    connect_feat(frequency_widget, inst, 'frequency')
    connect_feat(amplitude_widget, inst, 'amplitude')
    main.show()
    exit(app.exec_())
```

and that is all. Under the hood *connect_feat* is:

1.- Wrapping the widget to make it Lantz compatible.

2.- If applicable, configures minimum, maximum, steps and units.

3.- Add a handler such as when the widget value is changed, the Feat is updated.

4.- Add a handler such as when the Feat value is changed, the widget is updated.

The short way

If you have named the widgets according to the Feat name as we have done, you can save some typing (not so much here but a lot in big interfaces):

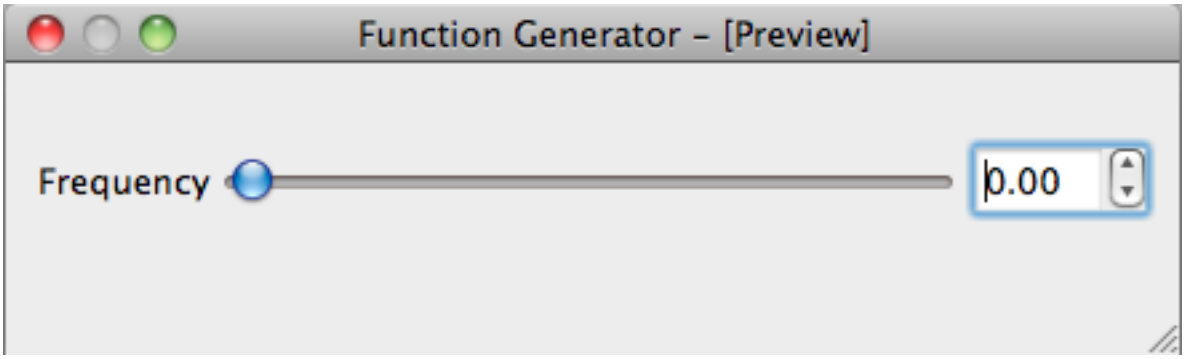
```
import sys

# Import lantz.ui register an import hook that will replace calls to Qt by PyQt4 or PySide ...
import lantz.ui

# and here we just use Qt and will work with both bindings!
from Qt.QtGui import QApplication, QWidget
from Qt.uic import loadUi

# From lantz we import the driver ...
from lantz.drivers.examples.fungen import LantzSignalGeneratorTCP

# and a function named connect_driver that does the work.
from lantz.ui.qtwidgets import connect_driver

app = QApplication(sys.argv)

# We load the UI from the QtDesigner file. You can also use pyuic4 to generate a class.
main = loadUi('connect_test.ui')

with LantzSignalGeneratorTCP('localhost', 5678) as inst:

    # We connect the parent widget (main) to the instrument.
    connect_driver(main, inst)
    main.show()
    exit(app.exec_())
```

Notice that now we do not need a reference to the widgets (only to the parent widget, here named main). And we call *connect_driver* (instead of *connect_feat*) without specifying the feat name. Under the hood, *connect_driver* is iterating over all widgets and checking if the driver contains a Feat with the widget name. If it does, it executes *connect_feat*.

See Also:

3.4.2 Connecting two (or more) widgets to the same feat

In many cases you want to have multiple widgets (e.g. different kind) connected to the same Feat. When the two widgets are together you could create a custom widget, but with Lantz it is not necessary. Consider the following UI:



You can set the frequency using the slider or the double spin box. The slider is named *frequency__slider* and the spin is named *frequency*.

The long way

You can connect each relevant driver Feat to the corresponding widget:

```
import sys

# Import lantz.ui register an import hook that will replace calls to Qt by PyQt4 or PySide ...
import lantz.ui

# and here we just use Qt and will work with both bindings!
from Qt.QtGui import QApplication, QWidget
from Qt.uic import loadUi

# From lantz we import the driver ...
from lantz.drivers.examples.fungen import LantzSignalGeneratorTCP

# and a function named connect_feat that does the work.
from lantz.ui.qtwidgets import connect_feat

app = QApplication(sys.argv)

# We load the UI from the QtDesigner file. You can also use pyuic4 to generate a class.
main = loadUi('connect_test.ui')

# We get a reference to each of the widgets.
slider = main.findChild((QWidget, ), 'frequency__slider')
spin = main.findChild((QWidget, ), 'frequency')

with LantzSignalGeneratorTCP('localhost', 5678) as inst:

    # We connect each widget to each feature
    # The syntax arguments are widget, target (driver), Feat name
```

```
    connect_feat(slider, inst, 'frequency')
    connect_feat(spin, inst, 'frequency')
    main.show()
    exit(app.exec_())
```

and that is all. Try it out and see how when you change one control the other one is updated.

The short way

If you have named the widgets according to the Feat and you have use a suffix in at least one of them, you can use *connect_driver*:

```
import sys

# Import lantz.ui register an import hook that will replace calls to Qt by PyQt4 or PySide ...
import lantz.ui

# and here we just use Qt and will work with both bindings!
from Qt.QtGui import QApplication, QWidget
from Qt.uic import loadUi

# From lantz we import the driver ...
from lantz.drivers.examples.fungen import LantzSignalGeneratorTCP

# and a function named connect_feat that does the work.
from lantz.ui.qtwidgets import connect_feat

app = QApplication(sys.argv)

# We load the UI from the QtDesigner file. You can also use pyuic4 to generate a class.
main = loadUi('connect_test.ui')

with LantzSignalGeneratorTCP('localhost', 5678) as inst:

    # We connect the parent widget (main) to the instrument.
    connect_driver(main, inst)
    main.show()
    exit(app.exec_())
```

Notice that now we do not need a reference to the widgets (only to the parent widget, here named main). And we call *connect_driver* (instead of *connect_feat*) without specifying the feat name. Under the hood, *connect_driver* is iterating over all widgets and checking if the driver contains a Feat with the widget name stripped from the suffix. If it does, it executes *connect_feat*.

In this example, we have use the double underscore __ to separate the suffix. This is a good choice and also the default as can be used in Qt and Python variable names. If you want have used another separator, you can specify it by passing the *sep* keyword argument:

```
connect_driver(main, inst, sep='_o_')
```

There is no limit in the number of widgets that you can connect to the same feat.

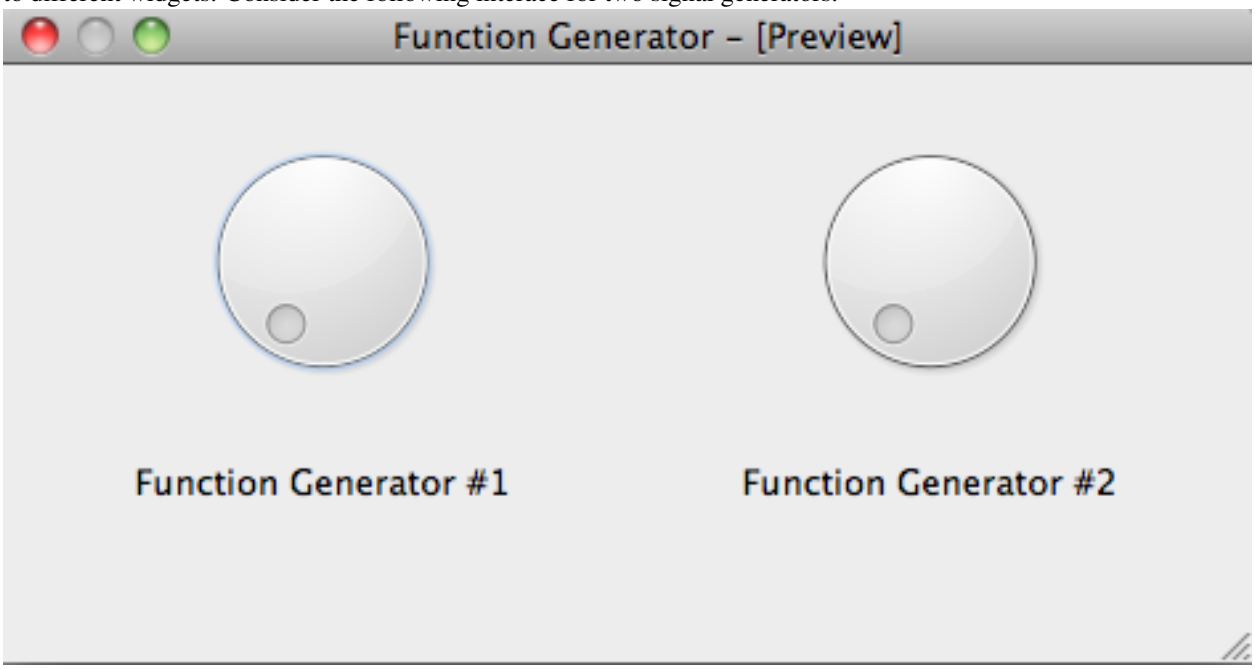
See Also:

Connecting a custom UI to a driver

Connecting two (or more) drivers

3.4.3 Connecting two (or more) drivers

Real application consists not only of a single instrument but many. In a custom UI, you can connect different drivers to different widgets. Consider the following interace for two signal generators.



(We use twice the same kind for simplicity, but it is not necessary).

The widgets are named *fungen1__frequency* and *fungen2__frequency*.

The long way

Get a reference to each widget and connect them manually:

```
import sys

# Import lantz.ui register an import hook that will replace calls to Qt by PyQt4 or PySide ...
import lantz.ui

# and here we just use Qt and will work with both bindings!
from Qt.QtGui import QApplication, QWidget
from Qt.uic import loadUi

# From lantz we import the driver ...
from lantz.drivers.examples.fungen import LantzSignalGeneratorTCP

# and a function named connect_feat that does the work.
from lantz.ui.qtwidgets import connect_feat

app = QApplication(sys.argv)

# We load the UI from the QtDesigner file. You can also use pyuic4 to generate a class.
main = loadUi('ui-two-drivers.ui')

# We get a reference to each of the widgets.
freq1 = main.findChild((QWidget, ), 'fungen1__frequency')
```

```
freq2 = main.findChild((QWidget, ), 'fungen2__frequency')

with LantzSignalGeneratorTCP('localhost', 5678) as inst1, \
     LantzSignalGeneratorTCP('localhost', 5679) as inst2:

    # We connect each widget to each feature
    # The syntax arguments are widget, target (driver), Feat name
    connect_feat(freq1, inst1, 'frequency')
    connect_feat(freq2, inst2, 'frequency')
    main.show()
    exit(app.exec_())
```

The not so long way

If you have use a prefix to solve the name collision you can use it and connect the driver:

```
import sys

# Import lantz.ui register an import hook that will replace calls to Qt by PyQt4 or PySide ...
import lantz.ui

# and here we just use Qt and will work with both bindings!
from Qt.QtGui import QApplication, QWidget
from Qt.uic import loadUi

# From lantz we import the driver ...
from lantz.drivers.examples.fungen import LantzSignalGeneratorTCP

# and a function named connect_feat that does the work.
from lantz.ui.qtwidgets import connect_feat

app = QApplication(sys.argv)

# We load the UI from the QtDesigner file. You can also use pyuic4 to generate a class.
main = loadUi('ui-two-drivers.ui')

with LantzSignalGeneratorTCP('localhost', 5678) as inst1, \
     LantzSignalGeneratorTCP('localhost', 5679) as inst2:

    # We connect each widget to each feature
    # The syntax arguments are widget, target (driver), Feat name
    connect_driver(main, inst1, prefix='fungen1')
    connect_driver(main, inst2, prefix='fungen1')
    main.show()
    exit(app.exec_())
```

This does not look like too much saving but if more than one Feat per driver to connect, *connect_driver* will do them all for you. Under the hood, *connect_driver* is iterating over all widgets and checking if the driver contains a Feat with the widget name prefixed by *prefix*. Note that we have used *fungen1* instead of *fungen1__* as the prefix. That is because *connect_driver* uses the double underscore as a separator by default. You can change it by passing the *sep* keyword argument.

The short way

If you have named the widgets according to the Feat name and added a prefix corresponding to the feat:

```
import sys

# Import lantz.ui register an import hook that will replace calls to Qt by PyQt4 or PySide ...
import lantz.ui

# and here we just use Qt and will work with both bindings!
from Qt.QtGui import QApplication, QWidget
from Qt.uic import loadUi

# From lantz we import the driver ...
from lantz.drivers.examples.fungen import LantzSignalGeneratorTCP

# and a function named connect_feat that does the work.
from lantz.ui.qtwidgets import connect_feat

app = QApplication(sys.argv)

# We load the UI from the QtDesigner file. You can also use pyuic4 to generate a class.
main = loadUi('ui-two-drivers.ui')

# Notice that now we specify the instrument name!
with LantzSignalGeneratorTCP('localhost', 5678, name='fungen1') as inst1, \
     LantzSignalGeneratorTCP('localhost', 5679, name='fungen2') as inst2:

    # We connect the whole main widget, and we give a list of drivers.
    connect_setup(main, [inst1, inst2])
    main.show()
    exit(app.exec_())
```

Under the hood, *connect_setup* iterates over all drivers in the second argument and executes *connect_driver* using the driver name.

See Also:

Connecting a custom UI to a driver

Connecting two (or more) widgets to the same feat

3.5 FAQs

3.5.1 Why building an instrumentation toolkit?

Instrumentation and experiment automation became a cornerstone of modern science. Most of the devices that we use to quantify and perturb natural processes can or should be computer controlled. Moreover, the ability to control and synchronize multiple devices, enables complex experiments to be accomplished in a reproducible manner.

This toolkit emerges from my frustration with existing languages, libraries and frameworks for instrumentation:

- Domain specific languages that make extremely difficult to achieve things that are trivial in most general-purpose languages.
- Lots of boilerplate code to achieve consistent behaviour across an application.
- Inability to use existing libraries.

Lantz aims to reduce the burden of writing a good instrumentation software by providing base classes from which you can derive your own. These classes provide the boilerplate code that enables advanced functionality, allowing you to concentrate in the program logic.

3.5.2 Why not using LabVIEW/LabWindows/Matlab?

LabVIEW is a development environment for a graphical programming language called "G" in which the flow of information in the program is determined by the connections between functions. While this concept is clear for non programmers, it quickly becomes a burden in big projects. Common procedures for source control, maintainable documentation, testing, and metaprogramming are cumbersome or just unavailable.

On the other hand, Matlab is a text based programming language with focus in numerical methods. It provides a set of additional function via its instrumentation toolbox.

Common to these two plataforms is that they have *evolved* a full fledged programming language from domain specific one while trying to maintain backwards compatibility. Many of the weird ways of doing things in these languages arise from this organic growth.

Unlike LabVIEW, LabWindows/CVI is ANSI C plus a set of convenient libraries for instrumentation. It brings all the goodies of C but it also all the difficulties such as memmory management.

Last but not least, these languages are propietary and expensive, locking your development. We need a free, open source toolkit for instrumentation build using a proven, mature, cross-plataform and well-though programming language.

3.5.3 But ... there are a lot of drivers already available for these languages

It is true, but many of these drivers are contributed by the users themselves. If a new toolkit emerges with enough momentum, many of those users will start to contribute to it. And due to the fact that building good drivers in Lantz is much easier than doing it in any of the other language we expect that this happens quite fast.

By the way, did you know we already have some *Drivers*. If your instrument is not listed, let us know!

3.5.4 Why Python?

Python is an interpreted, general-purpose high-level programming language. It combines a clear syntax, an excelent documentation and a large and comprehensive standard library. It is an awesome glue language that allows you to call already existing code in other languages. Finally, it is available in many platforms and is free.

3.5.5 Isn't Python slow?

Python is not slow. But even if it was, in instrumentation software the communication with the instrument is (by far) the rate limiting step. Sending a serial command that modifies the instrument function and receiving a response can easily take a few miliseconds and frequently much longer. While this might be fast in human terms, is an eternity for a computer. For this reason rapid prototyping, good coding practices and maintainability are more important for an instrumentation toolkit than speed.

3.5.6 But I do a lot of mathematical operations!

Slow operations such as numerical calculations are done using libraries such as NumPy and SciPy. This puts Python in the same line as Matlab and similar languages.

3.5.7 How do I start?

The *tutorial* is a good place.

3.5.8 I want to help. What can I do?

Please send comments and bug reports allowing us to make the code and documentation better to the issue tracker in GitHub

If you want to contribute with code, the drivers are a good place to start. If you have a programmed a new driver o improved an existing one, let us know.

If you have been using Lantz for a while, you can also write or clarify documentation helping people to use the toolkit.

The user interface also can use some help. We aim to provide widgets for common instrumentation scenarios.

Finally, talk to us if you have an idea that can be added to the core. We aim to keep the core small, robust and easy to maintain. However, patterns that appear recurrently when we work on drivers are factored out to the core after proven right.

Take a look at the *Contributing* section for more information.

3.5.9 Where does the name comes from?

It is a tribute to friend, Maximiliano Lantz. He was passionate scientist, teacher and science popularizer. We dreamt many times about having an instrumentation software simple to be used for teaching but powerful to be used for research. I hope that this toolkit fulfills these goals.

3.6 Drivers

3.6.1 lantz.drivers.aa

company AA Opto Electronic.

description Radio frequency and acousto-optic devices, Laser based sub-systems.

website http://opto.braggcell.com/

---

class `lantz.drivers.aa.`**`MDSnC`**(*port=1*, *timeout=1*, *write_timeout=1*, ***kwargs*)

Bases: `lantz.serial.SerialDriver`

MDSnC synthesizer for AOTF.nC

`CHANNELS = [0, 1, 2, 3, 4, 5, 6, 7]`

`enabled`

Keys ANY

`frequency`

Keys ANY

`main_enabled`

Values {False: 0, True: 1}

`power`

Keys ANY

Limits (0, 1023, 1)

powerdb

Keys ANY

3.6.2 lantz.drivers.aeroflex

company Aeroflex

description Test and measurement equipment and microelectronic solutions.

website http://www.aeroflex.com/

---

class lantz.drivers.aeroflex.**A2023a**(*port=1*, *timeout=1*, *write_timeout=1*, ***kwargs*)
Bases: lantz.serial.SerialDriver

Aeroflex Test Solutions 2023A 9 kHz to 1.2 GHz Signal Generator.

clear_status_async(**args*, ***kwargs*)

expose_async(**args*, ***kwargs*)

local_lockout(*value*)

remote(*value*)

reset_async(**args*, ***kwargs*)
(Async) Set the instrument functions to the factory default power up state.

self_test_async(**args*, ***kwargs*)
(Async) Is the interface and processor are operating?

software_handshake(*value*)

trigger_async(**args*, ***kwargs*)
(Async) Equivalent to Group Execute Trigger.

wait_async(**args*, ***kwargs*)
(Async) Inhibit execution of an overlapped command until the execution of the preceding operation has been completed.

RECV_TERMINATION = 256

SEND_TERMINATION = '\n'

amplitude

Units V

clear_status = functools.partial(<bound method Action.call of <lantz.action.Action object at 0x1a8a350>>, None)

event_status_enabled
Standard event enable register.

event_status_reg
Standard event enable register.

expose = functools.partial(<bound method Action.call of <lantz.action.Action object at 0x1a932f0>>, None)

fitted_options
Fitted options.

frequency

Units Hz

frequency_standard

Values {'INT', 'EXT10IND', 'EXTIND', 'EXT10DIR', 'INT10OUT'}

idn
Instrument identification.

offset

Units V

output_enabled

Values {False: 'DISABLED', True: 'ENABLED'}

phase

Units deg

reset = functools.partial(<bound method Action.call of <lantz.action.Action object at 0x1a82d30>>, None)

rflimit
Set RF output level max.

rflimit_enabled

Values {False: 'DISABLED', True: 'ENABLED'}

self_test = functools.partial(<bound method Action.call of <lantz.action.Action object at 0x1a82d90>>, None)

service_request_enabled
Service request enable register.

status_byte
Status byte, a number between 0-255.

time

Values {False: 'off', True: 'on'}

trigger = functools.partial(<bound method Action.call of <lantz.action.Action object at 0x1a82e50>>, None)

wait = functools.partial(<bound method Action.call of <lantz.action.Action object at 0x1a82df0>>, None)

3.6.3 lantz.drivers.andor

company Andor

description Scientific cameras.

website http://www.andor.com/

---

class lantz.drivers.andor.**Andor**(**args*, ***kwargs*)
Bases: lantz.foreign.LibraryDriver

close_async(**args*, ***kwargs*)
(Async) Close camera self.AT_H.

command(*strcommand*)
Run command.

finalize()
Finalise Library. Concluding function.

flush()

getbool(*strcommand*)
Run command and get Bool return value.

getenumerated(*strcommand*)
Run command and set Enumerated return value.

getfloat(*strcommand*)
Run command and get Int return value.

getint(*strcommand*)
Run command and get Int return value.

initialize()
Initialise Library.

is_implemented(*strcommand*)
Checks if command is implemented.

is_writable(*strcommand*)
Checks if command is writable.

open_async(**args*, ***kwargs*)
(Async) Open camera self.AT_H.

queuebuffer(*bufptr*, *value*)
Put buffer in queue.

setbool(*strcommand*, *value*)
Set command with Bool value parameter.

setenumerated(*strcommand*, *value*)
Set command with Enumerated value parameter.

setenumstring(*strcommand*, *item*)
Set command with EnumeratedString value parameter.

setfloat(*strcommand*, *value*)
Set command with Float value parameter.

setint(*strcommand*, *value*)
SetInt function.

waitbuffer(*ptr*, *bufsize*)
Wait for next buffer ready.

LIBRARY_NAME = 'atcore.dll'

close = functools.partial(<bound method Action.call of <lantz.action.Action object at 0x1a9d8f0>>, None)

open = functools.partial(<bound method Action.call of <lantz.action.Action object at 0x1a9a830>>, None)

class lantz.drivers.andor.**Neo**(**args*, ***kwargs*)
Bases: lantz.drivers.andor.andor.Andor

Neo Andor CMOS Camera

initialize()

take_image_async(**args*, ***kwargs*)
(Async) Image acquisition with circular buffer.

clock_rate

Values {200: '200 MHz', 280: '280 MHz', 100: '100 MHz'}

exposure_time
Get exposure time.

fan_peed
Fan speed.

pixel_encoding

Values {32: 'Mono32', 64: 'Mono64'}

roi
Set region of interest

sensor_size

sensor_temp
Sensor temperature.

take_image = functools.partial(<bound method Action.call of <lantz.action.Action object at 0x1aa3a10>>, None)

3.6.4 lantz.drivers.coherent

company Coherent Inc.

description Lasers and Lasers Systems.

website http://www.coherent.com/

---

copyright 2012 by Lantz Authors, see AUTHORS for more details.

license BSD, see LICENSE for more details.

class lantz.drivers.coherent.**Innova300C**(*port=1*, *baudrate=1200*, ***kwargs*)
Bases: lantz.serial.SerialDriver

Innova300 C Series.

center_powertrack_async(**args*, ***kwargs*)
(Async) Center PowerTrack and turn it off.

initialize()

query(*command*, *send_args*, *recv_args*)
Send query to the laser and return the answer, after handling possible errors.

Parameters

- **command** (*string*) – command to be sent to the instrument
- **send_args** – (termination, encoding) to override class defaults

- **recv_args** – (termination, encoding) to override class defaults

recalibrate_powertrack_async(**args*, ***kwargs*)
(Async) Recalibrate PowerTrack. This will only execute if PowerTrack is on and light regulation is off

ENCODING = 'ascii'

RECV_TERMINATION = '\r\n'

SEND_TERMINATION = '\r\n'

analog_enabled
Analog Interface input state.

Values {False: 0, True: 1}

analog_relative
Analog Interface input mode.

Values {False: 0, True: 1}

auto_light_cal_enabled
Automatic light regulation calibration flag.

Values {False: 0, True: 1}

autofill_delta
Tube voltage minus the autofill setting.

Units V

autofill_mode
Autofill mode.

Values {'disabled': 0, 'enabled until next autofill': 2, 'enabled': 1}

autofill_needed
Is the autofill needed (wheter fill is enabled or not)

Values {False: 0, True: 1}

baudrate

Values {19200, 9600, 300, 110, 1200, 2400, 4800}

cathode_current
Laser cathode current (AC).

Units A

cathode_voltage
Laser cathode voltage (AC).

Units V

center_powertrack = functools.partial(<bound method Action.call of <lantz.action.Action object at 0x1acfa50>>, No

control_pin_high
State of the input pin 10 of the Analog Interface.

Values {False: 0, True: 1}

current
Current regulation mode.

Units A

current_change_limit
Percent tube change before an automatic light regulation recalibration becomes necessary.

Limits (5, 100, 1)

current_range
Current corresponding to 5 Volts at the input or output lines of the Analog Interface.

Units A

Limits (10, 100, 1)

current_setpoint

Units A

Limits (0, 50, 0.01)

echo_enabled
Echo mode of the serial interface.

Values {False: 0, True: 1}

etalon_mode
Etalon mode.

Values {'modetune': 2, 'manual': 0, 'modetrack': 1}

etalon_temperature
Etalon temperature.

Units degC

etalon_temperature_setpoint

Units degC

Limits (51.5, 54, 0.001)

faults
List of all active faults.

head_software_rev
Software revision level in the laser head board.

idn
Laser identification, should be I300.

is_in_start_delay
Laser is in start delay (tube not ionized)

laser_enabled
Energize the power supply.

Values {False: 0, True: 2}

magnet_current
Laser magnet current.

Units A

magnetic_field_high
Magnetic field.

Values {False: 0, True: 1}

magnetic_field_setpoint_high

Values {False: 0, True: 1}

operating_mode
Laser operating mode.

Values {'current regulation': 0, 'standard light regulation': 2, 'reduced bandwidth light regulation': 1, 'current regulation, light regulation out of range': 3}

output_pin_high
State of the output pin 24 and 25 of the Analog Interface.

Values {(False, True): 2, (True, False): 1, (False, False): 0, (True, True): 3}

power
Current power output.

Units A

power_setpoint

Units W

Limits (0, 50, 0.0001)

powertrack_mode_enabled
PowerTrack.

Values {False: 0, True: 1}

powertrack_position

Keys ANY

Limits (0, 255)

recalibrate_powertrack = functools.partial(<bound method Action.call of <lantz.action.Action object at 0x1acfa30

remaining_time
Number of hours remaining before the laser will shut down automatically.

Units hour

software_rev
Software revision level in the power supply.

time_to_start
Timer countdown during the start delay cycle.

Units second

tube_time
Number of operating hours on the plasma tube.

Units hour

tube_voltage
Laser tube voltage.

Units V

water_flow
Water flow.

Units gallons/minute

water_resistivity
Resistivity of the incoming water to the power supply.

Units kohm*cm

water_temperature
Temperature of the incoming water to the power supply.

class lantz.drivers.coherent.**ArgonInnova300C**(*port=1*, *baudrate=1200*, ***kwargs*)
Bases: lantz.drivers.coherent.innova.Innova300C

Argon Innova 300C.

wavelength
Wavelength for the internal power meter calibration

Values {496, 514, 454, 488, 457, 364, 1090, 'MLUV', 528, 465, 'MLVS', 501, 472, 476, 'MLDUV', 351}

class lantz.drivers.coherent.**KryptonInnova300C**(*port=1*, *baudrate=1200*, ***kwargs*)
Bases: lantz.drivers.coherent.innova.Innova300C

Krypton Innova 300C.

wavelength
Wavelength for the internal power meter calibration

Values {482, 676, 647, 'MLVI', 'MLBG', 530, 'MLUV', 752, 520, 'MLVS', 'MLIR', 568, 'MLRD', 476}

3.6.5 lantz.drivers.examples

company Lantz Examples.

description Example drivers for simulated instruments.

website

---

class lantz.drivers.examples.**LantzSignalGenerator**
Bases: builtins.object

Lantz Signal Generator

query(*command*, *send_args*, *recv_args*)

ENCODING = 'ascii'

RECV_TERMINATION = '\n'

SEND_TERMINATION = '\n'

amplitude

Units V

Limits (10,)

calibrate = functools.partial(<bound method Action.call of <lantz.action.Action object at 0x1b179b0>>, None)

din

Keys ANY

Values {False: '0', True: '1'}

dout

Keys ANY

Values {False: '0', True: '1'}

frequency

Units Hz

Limits (1, 100000.0)

idn

offset

Units V

Limits (-5, 5, 0.01)

output_enabled

Values {False: 0, True: 1}

self_test = functools.partial(<bound method Action.call of <lantz.action.Action object at 0x1b179d0>>, None)

waveform

Values {'ramp': '3', 'sine': '0', 'square': '1', 'triangular': '2'}

class lantz.drivers.examples.**LantzSignalGeneratorTCP**(*host='localhost'*, *port=9997*, **args*, ***kwargs*)

Bases: lantz.drivers.examples.fungen.LantzSignalGenerator, lantz.network.TCPDriver

calibrate_async(**args*, ***kwargs*)
(Async) Calibrate.

self_test_async(**args*, ***kwargs*)
(Async) Reset to .

class lantz.drivers.examples.**LantzSignalGeneratorSerial**(*port=1*, *timeout=1*, *write_timeout=1*, ***kwargs*)

Bases: lantz.drivers.examples.fungen.LantzSignalGenerator, lantz.serial.SerialDriver

calibrate_async(**args*, ***kwargs*)
(Async) Calibrate.

self_test_async(**args*, ***kwargs*)
(Async) Reset to .

class lantz.drivers.examples.**LantzSignalGeneratorSerialVisa**(*resource_name*, **args*, ***kwargs*)

Bases: lantz.drivers.examples.fungen.LantzSignalGenerator, lantz.visa.SerialVisaDriver

calibrate_async(**args*, ***kwargs*)
(Async) Calibrate.

self_test_async(**args*, ***kwargs*)
(Async) Reset to .

class lantz.drivers.examples.**LantzVoltmeterTCP**(*host='localhost'*, *port=9997*, **args*, ***kwargs*)

Bases: lantz.network.TCPDriver

Lantz Signal Generator

auto_range_async(**args*, ***kwargs*)
(Async) Autoselect a range.

calibrate_async(**args*, ***kwargs*)
(Async) Calibrate.

query(*command*, *send_args*, *recv_args*)

self_test_async(**args*, ***kwargs*)
(Async) Self test

ENCODING = 'ascii'

RECV_TERMINATION = '\n'

SEND_TERMINATION = '\n'

auto_range = functools.partial(<bound method Action.call of <lantz.action.Action object at 0x1b24190>>, None)

calibrate = functools.partial(<bound method Action.call of <lantz.action.Action object at 0x1b241b0>>, None)

idn

range

Keys ANY

Values {1000: '4', 1: '1', 10: '2', 0.1: '0', 100: '3'}

self_test = functools.partial(<bound method Action.call of <lantz.action.Action object at 0x1b24210>>, None)

voltage

Keys ANY

Units V

3.6.6 lantz.drivers.kentech

company Kentech Instruments Ltd.

description Manufacturers of specialised and custom built electronics and imaging equipment.

website http://www.kentech.co.uk/

---

class lantz.drivers.kentech.**HRI**(*port=1*, *timeout=1*, *write_timeout=1*, ***kwargs*)
Bases: lantz.serial.SerialDriver

Kentech High Repetition Rate Image Intensifier.

clear_async(**args*, ***kwargs*)
(Async) Clear the buffer.

query(*command*, *send_args*, *recv_args*)
Send query to the instrument and return the answer. Set remote mode if needed.

query_expect(*command*, *recv_termination=None*, *expected='ok'*)

ENCODING = 'ascii'

RECV_TERMINATION = '\n'

SEND_TERMINATION = '\r'

average_voltage

Units volt

Limits (-50, 50)

clamp_voltage

Units volt

Limits (-50, 50)

clear = functools.partial(<bound method Action.call of <lantz.action.Action object at 0x1b24eb0>>, None)

enabled

Values {False, True}

mcp

Units volt

Limits (0, 1700)

mode

Values {'ldc': 22, 'inhibit': 0, 'hdc': 23, 'user3': 27, 'user1': 25, 'user2': 26, 'rf': 21, 'user4': 28, 'dc': 24}

remote

Values {False, True}

revision
Revision.

rfgain
RF Gain.

status
Get status.

temperature
Temperature.

trigger_ecl_level

Units centivolt

Limits (-40, 40, 1)

trigger_ecl_mode

Values {'log': 'LOGTRIG}', 'level': 'LVLTRIG'}

trigger_edge

Values {'falling': '-VETRIG}', 'rising': '+VETRIG'}

trigger_logic

Values {'ecl': 'ECLTRIG', 'ttl': 'TTLTRIG'}

trigger_ttl_termination

Values {'high': 'HITRIG', '50ohm': '50TRIG}'}

3.6.7 lantz.drivers.ni

company National Instruments

description

website http://www.ni.com/

3.6.8 lantz.drivers.olympus

company Olympus.

description Research and clinical microscopes.

website http://www.microscopy.olympus.eu/microscopes/

—

class `lantz.drivers.olympus.`**`IX2`**(*port=1*, *baudrate=19200*, *bytesize=8*, *parity='Even'*, *stopbits=1*, *flow=0*, *timeout=None*, *write_timeout=None*, **args*, ***kwargs*)

Bases: `lantz.drivers.olympus.ixbx.IXBX`

Olympus IX2 Body

`bottom_port_closed`
Bottom port

Values {False: 'OUT', True: 'IN'}

`camera_port_enabled`
Prism position

Values {False: '2', True: '1'}

`condensor`
Condensor position

Get procs

- <class 'int'>:set procs: - <class 'str'>

`filter_wheel`
Filter wheel position

Get procs

- <class 'int'>:set procs: - <class 'str'>

`mirror_unit`
Mirror unit position

Get procs

- <class 'int'>:set procs: - <class 'str'>

shutter1_closed
Shutter

Values {False: 'OUT', True: 'IN'}

shutter2_closed
Shutter

Values {False: 'OUT', True: 'IN'}

class lantz.drivers.olympus.**BX2A**(*port=1*, *baudrate=19200*, *bytesize=8*, *parity='Even'*, *stopbits=1*, *flow=0*, *timeout=None*, *write_timeout=None*, **args*, ***kwargs*)
Bases: lantz.drivers.olympus.ixbx.IXBX

Olympus BX2A Body

aperture_stop_diameter
Aperture stop diameter (DIA AS UCD)

Get procs

- <class 'int'>:set procs: - <class 'str'>

condenser_top_lens_enabled
Condenser top lens (UCD)

Values {False: 'OUT', True: 'IN'}

configure_filterwheel
Configure filterwheel

Get procs

- <class 'int'>:set procs: - <class 'str'>

cube
Cube position (RFAA/RLAA)

Get procs

- <class 'int'>:set procs: - <class 'str'>

shutter_closed
Shutter RFAA

Values {False: 'OUT', True: 'IN'}

turret
Turret position (UCD)

Get procs

- <class 'int'>:set procs: - <class 'str'>

class lantz.drivers.olympus.**IXBX**(*port=1*, *baudrate=19200*, *bytesize=8*, *parity='Even'*, *stopbits=1*, *flow=0*, *timeout=None*, *write_timeout=None*, **args*, ***kwargs*)
Bases: lantz.serial.SerialDriver

IX or BX Olympus microscope body.

init_origin()
Init origin

lamp_status()

move_relative_async(**args*, ***kwargs*)

query(*command*, *send_args*, *recv_args*)
Query the instrument and parse the response.

Raises InstrumentError

stop()
Stop any currently executing motion

RECV_TERMINATION = '\r\n'

SEND_TERMINATION = '\r\n'

body_locked
Turn the currently selected lamp on and off

Values {False: 'OFF', True: 'ON'}

fluo_shutter
External shutter for the fluorescent light source

Values {False: '0', True: '1'}

focus_locked
Turn the currently selected lamp on and off

Values {False: 'OFF', True: 'ON'}

idn
Microscope identification

jog_dial
Jog selection (Handle/BLA) ???

Values {False: 'FRM', True: 'FH'}

jog_enabled
Jog enabled

Values {False: 'OFF', True: 'ON'}

jog_limit_enabled
Jog limit enabled

Values {False: 'OFF', True: 'ON'}

jog_sensitivity
Jog sensitivity

Get procs

- <class 'int'>:set procs: - <class 'str'>

lamp_enabled
Turn the currently selected lamp onf and off

Values {False: 'OFF', True: 'ON'}

lamp_epi_enabled
Illumination source lamp.

Values {False: 'DIA', True: 'EPI'}

lamp_intensity
Transmitted light intensity

Get procs

- <class 'int'>:set procs: - <class 'str'>

move_relative = functools.partial(<bound method Action.call of <lantz.action.Action object at 0x1bcd290>>, None)

move_to_start_enabled
Sets / cancels returning operation to the start position after initializing the origin.

Values {False: 'OFF', True: 'ON'}

movement_status

objective
Objective nosepiece position

Get procs

- <class 'int'>:set procs: - <class 'str'>

soft_limits

Units (<Quantity(0.01, 'micrometer')>, <Quantity(0.01, 'micrometer')>)

z

Units 0.01 micrometer

3.6.9 lantz.drivers.pco

company PCO.

description High performance CCD-, CMOS- and sCMOS camera systems.

website http://www.pco.de

—

class lantz.drivers.pco.**Sensicam**(*board*, **args*, ***kwargs*)
Bases: lantz.foreign.LibraryDriver

PCO Sensicam

expose_async(**args*, ***kwargs*)
(Async) Expose.

Parameters **exposure** – exposure time.

finalize()

initialize()

read_out_async(**args*, ***kwargs*)
(Async) Readout image from the CCD.

Return type NumPy array

run_coc_async(**args*, ***kwargs*)

stop_coc_async(**args*, ***kwargs*)

take_image_async(**args*, ***kwargs*)
(Async) Take image.

Parameters **exposure** – exposure time.

Return type NumPy array

LIBRARY_NAME = 'senntcam.dll'

bel_time

Units microseconds

binning
Binning in pixels as a 2-element tuple (horizontal, vertical).

coc
Command Operation Code

coc_time

Units microseconds

delay_time

Units microseconds

exp_time

Units microseconds

expose = functools.partial(<bound method Action.call of <lantz.action.Action object at 0x1bec9d0>>, None)

exposure_time

Units ms

image_size
Image size in pixels (width, height).

image_status
Image status

mode
Imaging mode as a 3-element tuple (type, gain and submode).

read_out = functools.partial(<bound method Action.call of <lantz.action.Action object at 0x1becd30>>, None)

roi
Region of interest in pixels as a 4-element tuple (x1, x2, y1, y2).

run_coc = functools.partial(<bound method Action.call of <lantz.action.Action object at 0x1be3fb0>>, None)

status

stop_coc = functools.partial(<bound method Action.call of <lantz.action.Action object at 0x1be3f50>>, None)

table
COC table

take_image = functools.partial(<bound method Action.call of <lantz.action.Action object at 0x1becd70>>, None)

trigger
Triger mode.

3.6.10 lantz.drivers.rgblasersystems

company RGB Lasersysteme GmbH.

description Lasers and Lasers Systems.

website http://www.rgb-laser.com/

---

class `lantz.drivers.rgblasersystems.`**`MiniLasEvo`**(*port=1*, *timeout=1*, *write_timeout=1*, ***kwargs*)

Bases: `lantz.serial.SerialDriver`

Driver for any RGB Lasersystems MiniLas Evo laser.

`initialize`()

`query`(*command*, *send_args*, *recv_args*)

Send query to the laser and return the answer, after handling possible errors.

Parameters

- **command** (*string*) – command to be sent to the instrument
- **send_args** – (termination, encoding) to override class defaults
- **recv_args** – (termination, encoding) to override class defaults

`BAUDRATE` = 57600

`BYTESIZE` = 8

`DSRDTR` = False

`ENCODING` = 'ascii'

`PARITY` = 'none'

`RECV_TERMINATION` = '\r\n'

`RTSCTS` = False

`SEND_TERMINATION` = '\r\n'

`STOPBITS` = 1

`XONXOFF` = False

`available_features`

Available features (reserved for future use)

`control_mode`

Active current (power) control

`emission_wavelenght`

Emission wavelenght in nm

`enabled`

Values {False: '0', True: '1'}

`idn`

Identification of the device

`maximum_power`

Units mW

`operating_hours`

Total operating hours [hhhh:mm]

power

Units mW

software_version
Software version

status
Current device status

temperature
Current temperature in ºC

temperature_max
Highest operating temperature in ºC

temperature_min
Lowest operating temperature in ºC

3.6.11 lantz.drivers.sutter

company Sutter Instrument.

description Biomedical and scientific instrumentation.

website http://www.sutter.com/

—

class lantz.drivers.sutter.**Lambda103**(*port=11*, *baudrate=9600*, *timeout=1*, **args*, ***kwargs*)
Bases: lantz.serial.SerialDriver

High performance, microprocessor-controlled multi-filter wheel system for imaging applications requiring up to 3 filter wheels.

flush()
Flush.

motorsON_async(**args*, ***kwargs*)
(Async) Power on all motors.

reset_async(**args*, ***kwargs*)
(Async) Reset the controller.

status_async(**args*, ***kwargs*)

RECV_TERMINATION = ‘’

SEND_TERMINATION = ‘’

motorsON = functools.partial(<bound method Action.call of <lantz.action.Action object at 0x1c08610>>, None)

open_A

Values {False: ‘¬’, True: ‘ª’}

position

Keys ANY

remote

Values {False: 'ï', True: 'î'}

reset = functools.partial(<bound method Action.call of <lantz.action.Action object at 0x1c088b0>>, None)

status = functools.partial(<bound method Action.call of <lantz.action.Action object at 0x1c08690>>, None)

3.6.12 lantz.drivers.tektronix

company Tektronix.

description Test and Measurement Equipment.

website http://www.tek.com/

—

copyright 2012 by Lantz Authors, see AUTHORS for more details.

license BSD,

class lantz.drivers.tektronix.**TDS2024**(*port*)
Bases: lantz.visa.VisaDriver

Tektronix TDS2024 200 MHz 4 Channel Digital Real-Time Oscilloscope

initialize()
initiate.

acqparams = functools.partial(<bound method Action.call of <lantz.action.Action object at 0x1c11670>>, None)

autoconf = functools.partial(<bound method Action.call of <lantz.action.Action object at 0x1c11250>>, None)

curv = functools.partial(<bound method Action.call of <lantz.action.Action object at 0x1c11730>>, None)

dataencoding = functools.partial(<bound method Action.call of <lantz.action.Action object at 0x1c116d0>>, None)

datasource = functools.partial(<bound method Action.call of <lantz.action.Action object at 0x1c11610>>, None)

forcetrigger = functools.partial(<bound method Action.call of <lantz.action.Action object at 0x1c115b0>>, None)

idn
IDN.

measure_frequency = functools.partial(<bound method Action.call of <lantz.action.Action object at 0x1c11790>>, No

measure_max = functools.partial(<bound method Action.call of <lantz.action.Action object at 0x1c11850>>, None)

measure_mean = functools.partial(<bound method Action.call of <lantz.action.Action object at 0x1c118b0>>, None)

measure_min = functools.partial(<bound method Action.call of <lantz.action.Action object at 0x1c117f0>>, None)

trigger
Trigger state.

triggerlevel = functools.partial(<bound method Action.call of <lantz.action.Action object at 0x1c11550>>, None)

AA Opto Electronic.

- MDSnC synthesizer for AOTF.nC

Aeroflex

- Aeroflex Test Solutions 2023A 9 kHz to 1.2 GHz Signal Generator.

Andor

- Andor

- Neo Andor CMOS Camera

Coherent Inc.

- Argon Innova 300C.
- Innova300 C Series.
- Krypton Innova 300C.

Lantz Examples.

- LantzSignalGeneratorSerial
- LantzSignalGeneratorSerialVisa
- LantzSignalGeneratorTCP
- Lantz Signal Generator

Kentech Instruments Ltd.

- Kentech High Repetition Rate Image Intensifier.

National Instruments

Olympus.

- Olympus BX2A Body
- Olympus IX2 Body
- IX or BX Olympus microscope body.

PCO.

- PCO Sensicam

RGB Lasersysteme GmbH.

- Driver for any RGB Lasersystems MiniLas Evo laser.

Sutter Instrument.

- High performance, microprocessor-controlled multi-filter wheel system

Tektronix.

3.7 API

3.7.1 General

class lantz.**Driver**
Base class for all drivers.

Params name easy to remember identifier given to the instance for logging purposes

log(*level*, *msg*, **args*, ***kwargs*)
Log with the integer severity 'level' on the logger corresponding to this instrument.

Parameters

- **level** – severity level for this event.
- **msg** – message to be logged (can contain PEP3101 formatting codes)

log_critical(*msg*, **args*, ***kwargs*)
Log with the severity 'CRITICAL' on the logger corresponding to this instrument.

Parameters **msg** – message to be logged (can contain PEP3101 formatting codes)

log_debug(*msg*, **args*, ***kwargs*)
Log with the severity 'DEBUG' on the logger corresponding to this instrument.

Parameters **msg** – message to be logged (can contain PEP3101 formatting codes)

log_error(*msg*, **args*, ***kwargs*)
Log with the severity 'ERROR' on the logger corresponding to this instrument.

Parameters **msg** – message to be logged (can contain PEP3101 formatting codes)

log_info(*msg*, **args*, ***kwargs*)
Log with the severity 'INFO' on the logger corresponding to this instrument.

Parameters **msg** – message to be logged (can contain PEP3101 formatting codes)

log_warning(*msg*, **args*, ***kwargs*)
Log with the severity 'WARNING' on the logger corresponding to this instrument.

Parameters **msg** – message to be logged (can contain PEP3101 formatting codes)

recall(*keys=None*)
Return the last value seen for a feat or a collection of feats.

Parameters **keys** (*str, list, tuple, dict.*) – a string or list of strings with the properties to refresh. Default None all properties. If keys is a string, returns the value. If keys is a list, returns a dictionary.

refresh(*keys=None*)
Refresh cache by reading values from the instrument.

Parameters **keys** (*str or list or tuple or dict*) – a string or list of strings with the properties to refresh. Default None, meaning all properties. If keys is a string, returns the value. If keys is a list/tuple, returns a tuple. If keys is a dict, returns a dict.

refresh_async(*keys*, *callback=None*)
Asynchronous refresh cache by reading values from the instrument.

Parameters **keys** (*str or list or tuple or dict*) – a string or list of strings with the properties to refresh Default None, meaning all properties. If keys is a string, returns the value. If keys is a list, returns a dictionary.

Return type concurrent.future.

update(*newstate*, *force=None*, ***kwargs*)
Update driver.

Parameters

- **newstate** (*dict.*) – a dictionary containing the new driver state.
- **force** – apply change even when the cache says it is not necessary.
- **force** – boolean.

Raises ValueError if called with an empty dictionary.

update_async(*newstate*, *force*, *callback=None*, ***kwargs*)
Asynchronous update driver.

Parameters

- **newstate** (*dict.*) – driver state.
- **force** (*boolean.*) – apply change even when the cache says it is not necessary.
- **callback** (*callable.*) – Called when the update finishes.

Return type concurrent.future

Raises ValueError if called with an empty dictionary.

class lantz.**Feat** (*fget*, *fset*, *doc*, *values*, *units*, *limits=MISSING*, *procs=None*, *read_once=None*)
Pimped Python property for interfacing with instruments. Can be used as a decorator.

Processors can registered for each arguments to modify their values before they are passed to the body of the method. Two standard processors are defined: *values* and *units* and others can be given as callables in the *procs* parameter.

If a method contains multiple arguments, use a tuple. None can be used as *do not change*.

Parameters

- **fget** – getter function.
- **fset** – setter function.
- **doc** – docstring, if missing fget or fset docstring will be used.
- **values** – A dictionary to map key to values. A set to restrict the values. If a list/tuple instead of a dict is given, the value is not changed but only tested to belong to the container.
- **units** – *Quantity* or string that can be interpreted as units.
- **procs** – Other callables to be applied to input arguments.

modifiers = None
instance: key: value

value = None
instance: value

class lantz.**DictFeat** (*fget*, *fset=MISSING*, *doc=None*, *keys=None*, ***kwargs*)
Pimped Python property with getitem access for interfacing with instruments. Can be used as a decorator.

Takes the same parameters as *Feat*, plus:

Parameters **keys** – List/tuple restricts the keys to the specified ones.

class lantz.**Action** (*func*, *values*, *units*, *limits*, *procs=None*)
Wraps a Driver method with Lantz. Can be used as a decorator.

Processors can registered for each arguments to modify their values before they are passed to the body of the method. Two standard processors are defined: *values* and *units* and others can be given as callables in the *procs* parameter.

If a method contains multiple arguments, use a tuple. None can be used as *do not change*.

Parameters

- **func** – driver method to be wrapped.
- **values** – A dictionary to values key to values. If a list/tuple instead of a dict is given, the value is not changed but only tested to belong to the container.
- **units** – *Quantity* or string that can be interpreted as units.
- **procs** – Other callables to be applied to input arguments.

modifiers = None
instance: key: value

3.7.2 Interfacing to instruments

lantz.serial

Implements base classes for drivers that communicate with instruments via serial or parallel port.

copyright 2012 by Lantz Authors, see AUTHORS for more details.

license BSD, see LICENSE for more details.

class `lantz.serial.`**SerialDriver**(*port=1*, *timeout=1*, *write_timeout=1*, ***kwargs*)
Bases: `lantz.driver.TextualMixin`, `lantz.driver.Driver`

Base class for drivers that communicate with instruments via serial or parallel port using pyserial

Parameters

- **port** – Device name or port number
- **baudrate** – Baud rate such as 9600 or 115200
- **bytesize** – Number of data bits. Possible values = (5, 6, 7, 8)
- **parity** – Enable parity checking. Possible values = ('None', 'Even', 'Odd', 'Mark', 'Space')
- **stopbits** – Number of stop bits. Possible values = (1, 1.5, 2)
- **xonoff** – xonoff flow control enabled.
- **rtscts** – rtscts flow control enabled.
- **dsrdtr** – dsrdtr flow control enabled
- **timeout** – value in seconds, None or negative to wait for ever or 0 for non-blocking mode
- **write_timeout** – see timeout

finalize()
Close port

initialize()
Open port

raw_recv(*size*)
Receive raw bytes to the instrument.

Parameters **size** – number of bytes to receive

Returns received bytes

Return type bytes

If a timeout is set, it may return less bytes than requested. If size == -1, then the number of available bytes will be read.

raw_send(*data*)
Send raw bytes to the instrument.

Parameters

- **data** – bytes to be sent to the instrument
- **data** – bytes

BAUDRATE = 9600
communication parameters

RECV_CHUNK = -1
-1 is mapped to get the number of bytes pending.

RTSCTS = False
flow control flags

lantz.network

Implements a base class for drivers that communicate with instruments via TCP.

copyright 2012 by Lantz Authors, see AUTHORS for more details.

license BSD, see LICENSE for more details.

class `lantz.network.`**TCPDriver**(*host='localhost'*, *port=9997*, **args*, ***kwargs*)
Bases: `lantz.driver.TextualMixin`, `lantz.driver.Driver`

Base class for drivers that communicate with instruments via TCP.

Parameters

- **host** – Address of the network resource
- **port** – Port number

raw_recv(*size*)
Receive raw bytes to the instrument.

Parameters **size** – number of bytes to receive.

Returns received bytes.

Return type bytes.

raw_send(*data*)
Send raw bytes to the instrument.

Parameters

- **data** – bytes to be sent to the instrument.
- **data** – bytes.

lantz.foreign

Implements classes and methods to interface to foreign functions.

copyright 2012 by Lantz Authors, see AUTHORS for more details.

license BSD, see LICENSE for more details.

class `lantz.foreign.`**Library**(*library*, *prefix=''*, *wrapper=None*)
Bases: `builtins.object`

Library wrapper

Parameters

- **library** – ctypes library
- **wrapper** – callable that takes two arguments the name of the function and the function itself. It should return a callable.

class lantz.foreign.**LibraryDriver**(**args*, ***kwargs*)
Bases: lantz.driver.Driver

Base class for drivers that communicate with instruments calling a library (dll or others)

To use this class you must override LIBRARY_NAME

LIBRARY_NAME = ''
Name of the library

lantz.visa

Implements base classes for drivers that communicate with instruments using visalib.

copyright 2012 by Lantz Authors, see AUTHORS for more details.

license BSD, see LICENSE for more details.

class lantz.visa.**MessageVisaDriver**(*resource_name*, **args*, ***kwargs*)
Bases: lantz.driver.TextualMixin, lantz.driver.Driver

Base class for drivers that communicate with instruments via serial or parallel port using pyserial

Parameters **resource_name** – name or alias of the resource to open.

finalize()
Close port

initialize()
Open port

raw_send(*data*)
Send raw bytes to the instrument.

Parameters

- **data** – bytes to be sent to the instrument
- **data** – bytes

class lantz.visa.**SerialVisaDriver**(*resource_name*, **args*, ***kwargs*)
Bases: lantz.visa.MessageVisaDriver

Base class for drivers that communicate with instruments via serial port using visa.

Parameters **resource_name** – the visa resource name or alias (e.g. 'ASRL1::INSTR')

raw_recv(*size*)
Receive raw bytes to the instrument.

Parameters **size** – number of bytes to receive

Returns received bytes

Return type bytes

If a timeout is set, it may return less bytes than requested. If size == -1, then the number of available bytes will be read.

BAUDRATE = 9600
communication parameters

RTSCTS = False
flow control flags

3.7.3 UI

lantz.ui.qtwidgets

Implements UI widgets based on Qt widgets. To achieve functionality, instances of QtWidgets are patched.

copyright 2012 by Lantz Authors, see AUTHORS for more details.

license BSD, see LICENSE for more details.

class lantz.ui.qtwidgets.**ChildrenWidgets**(*parent*)
Convenience class to iterate children.

Parameters **parent** – parent widget.

class lantz.ui.qtwidgets.**DictFeatWidget**(*parent*, *target*, *feat*)
Widget to show a DictFeat.

Parameters

- **parent** – parent widget.
- **target** – driver object to connect.
- **feat** – DictFeat to connect.

setReadOnly(*value*)
Set read only s

setValue(*value*)
Set widget value.

value()
Get widget value.

lantz_target
Driver connected to this widget.

readable
If the Feat associated with the widget can be read (get).

writable
If the Feat associated with the widget can be written (set).

class lantz.ui.qtwidgets.**DriverTestWidget**(*parent*, *target*)
Widget that is automatically filled to control all Feats of a given driver.

Parameters

- **parent** – parent widget.
- **target** – driver object to map.

update_on_change(*new_state*)
Set the 'update_on_change' flag to new_state in each writable widget within this widget. If True, the driver will be updated after each change.

widgets_values_as_dict()
Return a dictionary mapping each writable feat name to the current value of the widget.

lantz_target
Driver connected to this widget.

class lantz.ui.qtwidgets.**FeatWidget**
Widget to show a Feat.

Parameters

- **parent** – parent widget.
- **target** – driver object to connect.
- **feat** – Feat to connect.

class lantz.ui.qtwidgets.**LabeledFeatWidget**(*parent*, *target*, *feat*)
Widget containing a label, a control, and a get a set button.

Parameters

- **parent** – parent widget.
- **target** – driver object to connect.
- **feat** – Feat to connect.

label_width
Width of the label

lantz_target
Driver connected to this widget.

readable
If the Feat associated with the widget can be read (get).

writable
If the Feat associated with the widget can be written (set).

class lantz.ui.qtwidgets.**SetupTestWidget**(*parent*, *targets*)
Widget to control multiple drivers.

Parameters

- **parent** – parent widget.
- **targets** – iterable of driver object to map.

class lantz.ui.qtwidgets.**UnitInputDialog**(*units*, *parent=None*)
Dialog to select new units. Checks compatibility while typing and does not allow to continue if incompatible.

Returns None if cancelled.

Parameters

- **units** – current units.
- **parent** – parent widget.

```
>>> new_units = UnitInputDialog.get_units('ms')
```

static get_units(*units*)
Creates and display a UnitInputDialog and return new units.

Return None if the user cancelled.

class lantz.ui.qtwidgets.**WidgetMixin**
Mixin class to provide extra functionality to QWidget derived controls.

Derived class must override _WRAPPED to indicate with which classes it can be mixed.

To wrap an existing widget object use:

```
>>> widget = QComboBox()
>>> WidgetMixin.wrap(widget)
```

If you want lantz to provide an appropriate wrapped widget for a given feat:

```
>>> widget = WidgetMixin.from_feat(feat)
```

In any case, after wrapping a widget you need to bind it to a feat:

```
>>> feat = driver.feats[feat_name]
>>> widget.bind_feat(feat)
```

Finally, you need to

```
>>> widget.lantz_target = driver
```

classmethod from_feat (*feat*, *parent=None*)
Return a widget appropriate to represent a lantz feature.

Parameters

- **feat** – a lantz feature proxy, the result of inst.feats[feat_name].
- **parent** – parent widget.

keyPressEvent (*event*)
When 'u' is pressed, request new units. When 'r' is pressed, get new value from the driver.

on_feat_value_changed (*value*, *old_value=MISSING*, *other=MISSING*)
When the driver value is changed, update the widget if necessary.

on_widget_value_changed (*value*, *old_value=MISSING*, *other=MISSING*)
When the widget is changed by the user, update the driver with the new value.

setReadOnly (*value*)
Set read only s

setValue (*value*)
Set widget value.

value ()
Get widget value.

value_from_feat ()
Update the widget value with the current Feat value of the driver.

value_to_feat ()
Update the Feat value of the driver with the widget value.

feat_key
Key associated with the DictFeat.

lantz_target
Driver connected to the widget.

readable
If the Feat associated with the widget can be read (get).

writable
If the Feat associated with the widget can be written (set).

lantz.ui.qtwidgets.**connect_driver** (*parent*, *target*, *prefix*, *sep*)
Connect all children widgets to their corresponding lantz feature matching by name. Non-matching names are ignored.

Parameters

- **parent** – parent widget.

- **target** – the driver.
- **prefix** – prefix to be prepended to the lantz feature (default = '')
- **sep** – separator between prefix, name and suffix

lantz.ui.qtwidgets.**connect_feat**(*widget*, *target*, *feat_name=None*, *feat_key=MISSING*)
Connect a feature from a given driver to a widget. Calling this function also patches the widget is necessary.

If applied two times with the same widget, it will connect to the target provided in the second call. This behaviour can be useful to change the connection target without rebuilding the whole UI. Alternative, after connect has been called the first time, widget will have a property *lantz_target* that can be used to achieve the same thing.

Parameters

- **widget** – widget instance.
- **target** – driver instance.
- **feat_name** – feature name. If None, connect using widget name.
- **feat_key** – For a DictFeat, this defines which key to show.

lantz.ui.qtwidgets.**connect_setup**(*parent*, *targets*, *prefix*, *sep*)
Connect all children widget to their corresponding

Parameters

- **parent** – parent widget.
- **targets** – iterable of drivers.
- **prefix** – prefix to be prepended to the lantz feature name if None, the driver name will be used (default) if it is a dict, the driver name will be used to obtain he prefix.

lantz.ui.qtwidgets.**register_wrapper**(*cls*)
Register a class as lantz wrapper for QWidget subclasses.

The class must contain a field (_WRAPPERS) with a tuple of the QWidget subclasses that it wraps.

lantz.ui.qtwidgets.**request_new_units**(*current_units*)
Ask for new units using a dialog box and return them.

Parameters **current_units** (*Quantity*) – current units or magnitude.

lantz.ui.qtwidgets.**start_test_app**(*target*, *width=500*, **args*)
Start a single window test application with a form automatically generated for the driver.

Parameters

- **target** – a driver object or a collection of drivers.
- **width** – to be used as minimum width of the window.
- **args** – arguments to be passed to QApplication.

3.7.4 Support

lantz.stats

Implements an statistical accumulator

copyright 2012 by Lantz Authors, see AUTHORS for more details.

license BSD, see LICENSE for more details.

class lantz.stats.**RunningState**(*value=None*)
Accumulator for events.

Parameters **value** – first value to add.

add(*value*)
Add to the accumulator.

Parameters **value** – value to be added.

class lantz.stats.**RunningStats**
Accumulator for categorized event statistics.

add(*key*, *value*)
Add an event to a given accumulator.

Parameters

- **key** – category to which the event should be added.
- **value** – value of the event.

stats(*key*)
Return the statistics for the current accumulator.

Return type Stats.

class lantz.stats.**Stats**
Data structure

count
Alias for field number 1

last
Alias for field number 0

max
Alias for field number 5

mean
Alias for field number 2

min
Alias for field number 4

std
Alias for field number 3

lantz.stats.**stats**(*state*)
Return the statistics for given state.

Parameters **state** (*RunningState*) – state

Returns statistics

Return type Stats named tuple

lantz.processors

copyright 2012 by Lantz Authors, see AUTHORS for more details.

license BSD, see LICENSE for more details.

class lantz.processors.**FromQuantityProcessor**
Processor to convert the units the function arguments.

The syntax is equal to *Processor* except that strings are interpreted as units.

```
>>> conv = FromQuantityProcessor('ms')
>>> conv(Q_(1, 's'))
1000.0
```

class lantz.processors.**MapProcessor**
Processor to map the function parameter values.

The syntax is equal to *Processor* except that a dict is used as mapping table.

Examples:

```
>>> conv = MapProcessor({True: 42})
>>> conv(True)
42
```

class lantz.processors.**ParseProcessor**
Processor to convert/parse the function parameters.

The syntax is equal to *Processor* except that strings are interpreted as a :class:Parser expression.

```
>>> conv = ParseProcessor('spam {:s} eggs')
>>> conv('spam ham eggs')
'ham'

>>> conv = ParseProcessor(('hi {:d}', 'bye {:s}'))
>>> conv(('hi 42', 'bye Brian'))
(42, 'Brian')
```

class lantz.processors.**Processor**
Processor to convert the function parameters.

A *callable* argument will be used to convert the corresponding function argument.

For example, here *x* will be converted to float, before entering the function body:

```
>>> conv = Processor(float)
>>> conv
<class 'float'>
>>> conv('10')
10.0
```

The processor supports multiple argument conversion in a tuple:

```
>>> conv = Processor((float, str))
>>> type(conv)
<class 'lantz.processors.Processor'>
>>> conv(('10', 10))
(10.0, '10')
```

class lantz.processors.**RangeProcessor**
Processor to convert the units the function arguments.

The syntax is equal to *Processor* except that iterables are interpreted as (low, high, step) specified ranges. Step is optional and max is included

```
>>> conv = RangeProcessor(((1, 2, .5), ))
>>> conv(1.7)
1.5
```

class lantz.processors.**ReverseMapProcessor**
Processor to map the function parameter values.

The syntax is equal to *Processor* except that a dict is used as mapping table.

Examples:

```
>>> conv = ReverseMapProcessor({True: 42})
>>> conv(42)
True
```

class lantz.processors.**ToQuantityProcessor**
Decorator to convert the units the function arguments.

The syntax is equal to *Processor* except that strings are interpreted as units.

```
>>> conv = ToQuantityProcessor('ms')
>>> conv(Q_(1, 's'))
<Quantity(1000.0, 'millisecond')>
>>> conv(1)
<Quantity(1.0, 'millisecond')>
```

lantz.processors.**check_membership**(*container*)

Parameters **container** –

Returns

```
>>> checker = check_membership((1, 2, 3))
>>> checker(1)
1
>>> checker(0)
Traceback (most recent call last):
...
ValueError: 0 not in (1, 2, 3)
```

lantz.processors.**check_range_and_coerce_step**(*low*, *high*, *step=None*)

Parameters

- **low** –
- **high** –
- **step** –

Returns

```
>>> checker = check_range_and_coerce_step(1, 10)
>>> checker(1), checker(5), checker(10)
(1, 5, 10)
>>> checker(11)
Traceback (most recent call last):
...
ValueError: 11 not in range (1, 10)
>>> checker = check_range_and_coerce_step(1, 10, 1)
>>> checker(1), checker(5.4), checker(10)
(1, 5, 10)
```

lantz.processors.**convert_to**(*units*, *on_dimensionless='warn'*, *on_incompatible='raise'*, *return_float=False*)
Return a function that convert a Quantity to to another units.

Parameters

- **units** – string or Quantity specifying the target units
- **on_dimensionless** – how to proceed when a dimensionless number number is given. 'raise' to raise an exception, 'warn' to log a warning and proceed, 'ignore' to silently proceed
- **on_incompatible** – how to proceed when source and target units are incompatible. Same options as *on_dimensionless*

Raises

ValueError if the incoming value cannot be properly converted

```
>>> convert_to('mV')(Q_(1, 'V'))
<Quantity(1000.0, 'millivolt')>
>>> convert_to('mV', return_float=True)(Q_(1, 'V'))
1000.0
```

lantz.processors.**get_mapping**(*container*)

```
>>> getter = get_mapping({'A': 42, 'B': 43})
>>> getter('A')
42
>>> checker(0)
Traceback (most recent call last):
...
ValueError: 0 not in ('A', 'B')
```

lantz.processors.**getitem**(*a*, *b*)
Return a[b] or if not found a[type(b)]

lantz.visalib

Wraps Visa Library in a Python friendly way.

This wrapper originated while porting pyvisa to Python 3 and therefore is heavily influenced by it. There are a few important differences:

- There is no visa_library singleton object and the library path can be specified.
- Similar functions for different data width (In8, In16, etc) have been grouped within the same function. The extended versions are also grouped.
- VISA functions dealing with strings have been dropped as can be easily replaced by Python functions.
- types, status codes, attributes, events and constants are defined within a class (not a module).
- Prefixes in types (*vi*), status codes (*VI_*), attributes (*VI_ATTR*), events (*VI_EVENT*) and constants (*VI_*) have been dropped for clarity. As this constants are defined within a RichEnum class, prefixed names are still usable.

class lantz.visalib.**Attributes**
Enumeration of VISA Attributes with their corresponding value, VISA TYPE and docstring

class lantz.visalib.**Constants**
Enumeration of VISA Constants with their corresponding value and docstring

class lantz.visalib.**Events**
Enumeration of VISA Events with their corresponding value and docstring

class lantz.visalib.**ResourceInfo**
Resource extended information

alias
Alias for field number 4

interface_board_number
Alias for field number 1

interface_type
Alias for field number 0

resource_class
Alias for field number 2

resource_name
Alias for field number 3

class lantz.visalib.**ResourceManager**
VISA Resource Manager

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

list_resources(*query='?*::INSTR'*)
Returns a list 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.

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.

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

REGISTER = {}
Holds a mapping between library_path and the default manager

class lantz.visalib.**RichEnum**
Type for rich enumerations.

class lantz.visalib.**StatusCode**
Enumeration of VISA status codes with their corresponding value and docstring.

class lantz.visalib.**Types**
Enumeration of VISA types mapped to ctypes.

class lantz.visalib.**VisaLibrary**
VISA Library wrapper.

Parameters **library_path** – full path of the library. If not given, the default value LIBRARY_PATH it is used.

assert_interrupt_signal(*session*, *mode*, *status_id*)
Asserts the specified interrupt or signal.

Parameters

- **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(*session*, *protocol*)
Asserts software or hardware trigger.

Parameters

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

assert_utility_signal(*session*, *line*)
Asserts or deasserts the specified utility bus signal.

Parameters

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

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

Parameters

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

Returns data read.

Return type bytes

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

Parameters

- **session** – Unique logical identifier to a session.
- **data** – data to be written.

Returns number of written bytes.

clear(*session*)
Clears a device.

Parameters **session** – Unique logical identifier to a session.

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

Parameters **session** – Unique logical identifier to a session, event, or find list.

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

Parameters

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

Parameters

- **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 (*session*, *event_type*, *mechanism*, *context=0*)
Discards event occurrences for specified event types and mechanisms in a session.

Parameters

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

Parameters **find_list** – Describes a find list. This parameter must be created by find_resources().

Returns Returns a string identifying the location of a device.

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

Parameters

- **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

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

Parameters

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

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

Parameters

- **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.

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

Parameters

- **session** – Unique logical identifier to a session.
- **data** – data tor write.

Returns Number of written bytes.

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

Parameters

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

Parameters

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

Parameters

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

Parameters **session** – Unique logical identifier to a session.

Returns

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.

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

Parameters

- **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 (*session*, *map_space*, *map_base*, *map_size*, *access=0*, *suggested=0*, *extended=False*)
Maps the specified memory space into the process's address space.

Parameters

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

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

map_trigger (*session*, *trigger_source*, *trigger_destination*, *mode=0*)
Map the specified trigger source line to the specified destination line.

Parameters

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

Parameters

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

Parameters

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

move(*session*, *source_space*, *source_offset*, *source_width*, *destination_space*, *destination_offset*, *destination_width*, *length*, *extended=False*)
Moves a block of data.

Parameters

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

move_async(*session*, *source_space*, *source_offset*, *source_width*, *destination_space*, *destination_offset*, *destination_width*, *length*, *extended=False*)
Moves a block of data asynchronously.

Parameters

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

Returns Job identifier of this asynchronous move operation.

move_memory_in(*session*, *space*, *offset*, *length*, *width*, *extended=False*)
Moves a block of data from the specified address space and offset to local memory.

Parameters

- **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.
- **width** – Number of bits to read per element.
- **extended** – Use 64 bits offset independent of the platform.

Returns Data read from bus.

Corresponds to viIn* functions of the visa library.

move_memory_out(*session*, *space*, *offset*, *length*, *data*, *width*, *extended=False*)
Moves a block of data from local memory to the specified address space and offset.

Parameters

- **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.
- **width** – Number of bits to read per element.
- **extended** – Use 64 bits offset independent of the platform.

open(*session*, *resource_name*, *access_mode=0*, *open_timeout=0*)
Opens a session to the specified resource.

Parameters

- **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()
This function returns a session to the Default Resource Manager resource.

Returns Unique logical identifier to a Default Resource Manager session.

parse_resource(*session*, *resource_name*)
Parse a resource string to get the interface information.

Parameters

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

Parameters

- **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(*session*, *address*, *width*)
Writes an 8-bit, 16-bit, 32-bit, or 64-bit value from the specified address.

Parameters

- **session** – Unique logical identifier to a session.
- **address** – Source address to read the value.
- **width** – Number of bits to read.

Returns Data read from bus.

Return type bytes

poke(*session*, *address*, *data*, *width*)
Reads an 8-bit, 16-bit, 32-bit, or 64-bit value from the specified address.

Parameters

- **session** – Unique logical identifier to a session.
- **address** – Source address to read the value.
- **data** – value to be written to the bus.
- **width** – Number of bits to read.

Returns Data read from bus.

Return type bytes

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

Parameters

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

Returns data read.

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

Parameters

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

Returns (ctypes buffer with result, jobid)

read_memory(*session*, *space*, *offset*, *width*, *extended=False*)
Reads in an 8-bit, 16-bit, 32-bit, or 64-bit value from the specified memory space and offset. :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 width: Number of bits to read. :param extended: Use 64 bits offset independent of the platform. :return: Data read from memory.

Corresponds to viIn* functions of the visa library.

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

Parameters **session** – Unique logical identifier to a session.

Returns Service request status byte.

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

Parameters

- **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.

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

Parameters

- **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.

Returns

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

Parameters

- **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).

Returns

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

Parameters

- **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.

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

Parameters

- **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(*session*)
Relinquishes a lock for the specified resource.

Parameters **session** – Unique logical identifier to a session.

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

Parameters **session** – Unique logical identifier to a session.

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

Parameters

- **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*)

Returns

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

Parameters

- **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(*session*, *request_type_bitmap_field*, *request_id*, *request_value*, *index*, *data=''*)
Performs a USB control pipe transfer to the device.

Parameters

- **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(*session*, *mode*, *command*)
Sends the device a miscellaneous command or query and/or retrieves the response to a previous query.

Parameters

- **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(*session*, *in_event_type*, *timeout*)
Waits for an occurrence of the specified event for a given session.

Parameters

- **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(*session*, *data*)
Writes data to device or interface synchronously.

Parameters

- **session** – Unique logical identifier to a session.
- **data** – data to be written.

Returns Number of bytes actually transferred.

write_asynchronously(*session*, *buffer*)
Writes data to device or interface asynchronously.

Parameters

- **session** – Unique logical identifier to a session.
- **data** – data to be written.

Returns Job ID of this asynchronous write operation.

write_from_file(*session*, *filename*, *count*)
Take data from a file and write it out synchronously.

Parameters

- **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.

write_memory(*session*, *space*, *offset*, *data*, *width*, *extended=False*)
Reads in an 8-bit, 16-bit, 32-bit, or 64-bit value from the specified memory space and offset. :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 width: Number of bits to read. :param extended: Use 64 bits offset independent of the platform.

Corresponds to viOut* functions of the visa library.

REGISTER = {}
Holds a mapping between library_path and VisaLibrary objects

lantz.stringparser

A stand alone module used by lantz. (website)

Motivation

The `stringparser` module provides a simple way to match patterns and extract information within strings. As patterns are given using the familiar format string specification **PEP 3101**, writing them is much easier than writing regular expressions (albeit less powerful).

Examples

You can build a reusable parser object:

```
>>> parser = Parser('The answer is {:d}')
>>> parser('The answer is 42')
42
>>> parser('The answer is 54')
54
```

Or directly:

```
>>> Parser('The answer is {:d}')('The answer is 42')
42
```

You can retrieve many fields:

```
>>> Parser('The {:s} is {:d}')('The answer is 42')
('answer', 42)
```

And you can use numbered fields to order the returned tuple:

```
>>> Parser('The {1:s} is {0:d}')('The answer is 42')
(42, 'answer')
```

Or named fields to return an OrderedDict:

```
>>> Parser('The {a:s} is {b:d}')('The answer is 42')
OrderedDict([('a', 'answer'), ('b', 42)])
```

You can ignore some fields using _ as a name:

```
>>> Parser('The {_:s} is {:d}')('The answer is 42')
42
```

Limitations

- From the format string: *[[fill]align][sign][#][0][minimumwidth][.precision][type]* only *type*, *sign* and *#* are currently implemented. This might cause trouble to match certain notation like:
  - decimal: '-4' written as '- 4'
  - etc
- Lines are matched from beginning to end. {:d} will NOT return all the numbers in the string. Use regex for that.

3.8 Contributing

You are most welcome to contribute to Lantz with code, documentation and translations. Please read the following document for guidelines.

3.8.1 Python style

- Unless otherwise specified, follow **PEP 8** strictly.
- Document every class and method according to **PEP 257**.

- Before submitting your code, use a tool like pep8.py and pylint.py to check for style.
- *Feat* and *DictFeat* should be named with a noun or an adjective.
- *Action* should be named with a verb.
- Files should be utf-8 formatted.

3.8.2 Header

All files must have first the encoding indication, and then a header indicating the module, a small description and the copyright message. For example:

```
# -*- coding: utf-8 -*-
"""
    lantz.foreign
    ~~~~~~~~~~~~~

    Implements classes and methods to interface to foreign functions.

    :copyright: (c) 2012 by Lantz Authors, see AUTHORS for more details.
    :license: BSD, see LICENSE for more details.
"""
```

3.8.3 Copyright

Files in the Lantz repository don't list author names, both to avoid clutter and to avoid having to keep the lists up to date. Instead, your name will appear in the Git change log and in the AUTHORS file. The Lantz maintainer will update this file when you have submitted your first commit.

Before your first contribution you must submit the *Contributor Agreement*. Code that you contribute should use the standard copyright header:

```
:copyright: (c) 2012 by Lantz Authors, see AUTHORS for more details.
:license: BSD, see LICENSE for more details.
```

3.8.4 Finally, we have a small Zen

```
import this
Lantz should not get in your way.
Unless you actually want it to.
Even then, python ways should not be void.
Provide solutions for common scenarios.
Leave the special cases for the people who actually need them.
Logging is great, do it often!
```

The easiest way is to start *Contributing Drivers*. Once that you gain experience with *Lantz* you can start *Contributing to the core*.

3.9 Contributing Drivers

The most straightforward way to contribute to Lantz is by submitting instrument drivers. You do not need to clone or understand the whole structure of *Lantz* for this purpose and you can do it directly from you own projects.

If you have installed Lantz using the tutorial (*installing*), you

    $ lantz-contribute <filename>

Please be sure that you have documented the code properly before submission. You can also use this tool if you want some feedback about your code.

3.10 Contributing to the core

To contribute to the core, you need to clone the *Lantz* repository first.

3.10.1 Version control system

Lantz uses Git as version control system.

There are always at least two branches:

- master: appropriate for users. It must always be in a working state.
- develop: appropriate for developers. Might not be in a working state.

The master branch only accepts atomic, small commits. Larger changes that might break the master branch should happen in the develop branch. The develop branch will be merged into the master after deep testing. If you want to refactor major parts of the code or try new ideas, create a dedicated branch. This branch will merged into develop once tested.

The easiest way to start hacking Lantz codebase is using a virtual environment and cloning an editable package.

Assuming that you have installed all the requirements described in *Installation guide*, in OSX/Linux:

```
$ pip-3.2 install virtualenv
$ cd ~
$ virtualenv -p python3.2 --system-site-packages lantzenv
$ cd lantzenv
$ source bin/activate
```

and in Windows:

```
C:\Python3.2\Scripts\pip install virtualenv
cd  %USERPROFILE%\Desktop
C:\Python32\Scripts\virtualenv --system-site-packages lantzenv
cd lantzenv\Scripts
activate
```

and then install an editable package:

```
$ pip install -e git+gitolite@glugcen.dc.uba.ar:lantz.git#egg=lantz
```

or from **'Lantz at Github'_**:

```
$ pip install -e git+git://github.com/hgrecco/lantz.git#egg=lantz
```

You will find the code in *~/lantzenv/src/lantz* (OSX/Linux) or *%USERPROFILE%\Desktop\lantzenv\src\lantz* (Windows).

3.10.2 File system structure

The distribution is organized in the following folders:

docs

Documentation in reStructuredText format with Sphinx makefile. Files must have a `.rst` extension

To generate, for example, HTML documentation change into this folder and run:

```
$ make html
```

You will find the generated documentation in `docs/_build/html/index.html`

examples

Root folder for the examples.

lantz

Root folder containing the core functionality

drivers

There is a package folder for each manufacturer and module file for each instrument model (or family of models). All files are named using lowercase. Class drivers are named according to the model. If the model starts with a number, then the first letter of the manufacturer should be prefixed. Finally, all classes should be imported in the __init__.py of the corresponding package.

simulators

Instrument simulators

ui

User interface related code.

scripts

Python scripts to provide simple command line functionality.

tests

Test cases.

3.10.3 Python style

- Unless otherwise specified, follow **PEP 8** strictly.
- Document every class and method according to **PEP 257**.
- Before submitting your code, use a tool like pep8.py and pylint.py to check for style.
- *Feat* and *DictFeat* should be named with a noun or an adjective.
- *Action* should be named with a verb.
- Files should be utf-8 formatted.

3.10.4 Header

All files must have first the encoding indication, and then a header indicating the module, a small description and the copyright message. For example:

```
# -*- coding: utf-8 -*-
"""
    lantz.foreign
    ~~~~~~~~~~~~~

    Implements classes and methods to interface to foreign functions.

    :copyright: (c) 2012 by Lantz Authors, see AUTHORS for more details.
    :license: BSD, see LICENSE for more details.
"""
```

3.10.5 Submitting your changes

Changes must be submitted for merging as patches or pull requests.

Before doing so, please check that:

- The new code is functional.
- The new code follows the style guidelines.
- The new code is documented.
- All tests are passed.
- Any new file contains an appropriate header.
- You commit to the head of the appropriate branch (usually develop).

Commits must include a one-line description of the intended change followed, if necessary, by an empty line and detailed description. You can send your patch by e-mail to *lantz.contributor@gmail.com*:

```
$ git format-patch origin/develop..develop
0001-Changed-Driver-class-to-enable-inheritance-of-Action.patch
0002-Added-RECV_CHUNK-to-TextualMixin.patch
```

or send a pull request.

3.10.6 Copyright

Files in the Lantz repository don't list author names, both to avoid clutter and to avoid having to keep the lists up to date. Instead, your name will appear in the Git change log and in the AUTHORS file. The Lantz maintainer will update this file when you have submitted your first commit.

Before your first contribution you must submit the *Contributor Agreement*. Code that you contribute should use the standard copyright header:

```
:copyright: (c) 2012 by Lantz Authors, see AUTHORS for more details.
:license: BSD, see LICENSE for more details.
```

3.10.7 Finally, we have a small Zen

```
import this
Lantz should not get in your way.
Unless you actually want it to.
Even then, python ways should not be void.
Provide solutions for common scenarios.
Leave the special cases for the people who actually need them.
Logging is great, do it often!
```

3.11 Community

The official **mailing list**, hosted in GlugCEN, is *lantz@glugcen.dc.uba.ar* and is used for bug reports and general discussions. You can subscribe in GlugCEN lantz.

There is an additional mailing list in Spanish, *lantz-ar@glugcen.dc.uba.ar*, mostly for local activities in Argentina but also for general discussion and support. You can subscribe in GlugCEN lantz-ar.

You can report bugs, as well as request new features in the **issue tracker** on GitHub.

3.12 Reporting Bugs

If you have found any error in the code or the documentation, please report it using GitHub issue tracker.

To make you bug report as useful as possible, please add a comprehensive description of what you were trying to do, what you have observed, and what you expected.

Additionally if you have a patch, feel free to contribute it back. Check on *Contributing* for more information.

We thank GlugCEN for hosting the code, the docs and the mailing list

PYTHON MODULE INDEX

d

f

n

p

s

u

v

INDEX

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z