Lantz Documentation

Release 0.3

Hernán E. Grecco

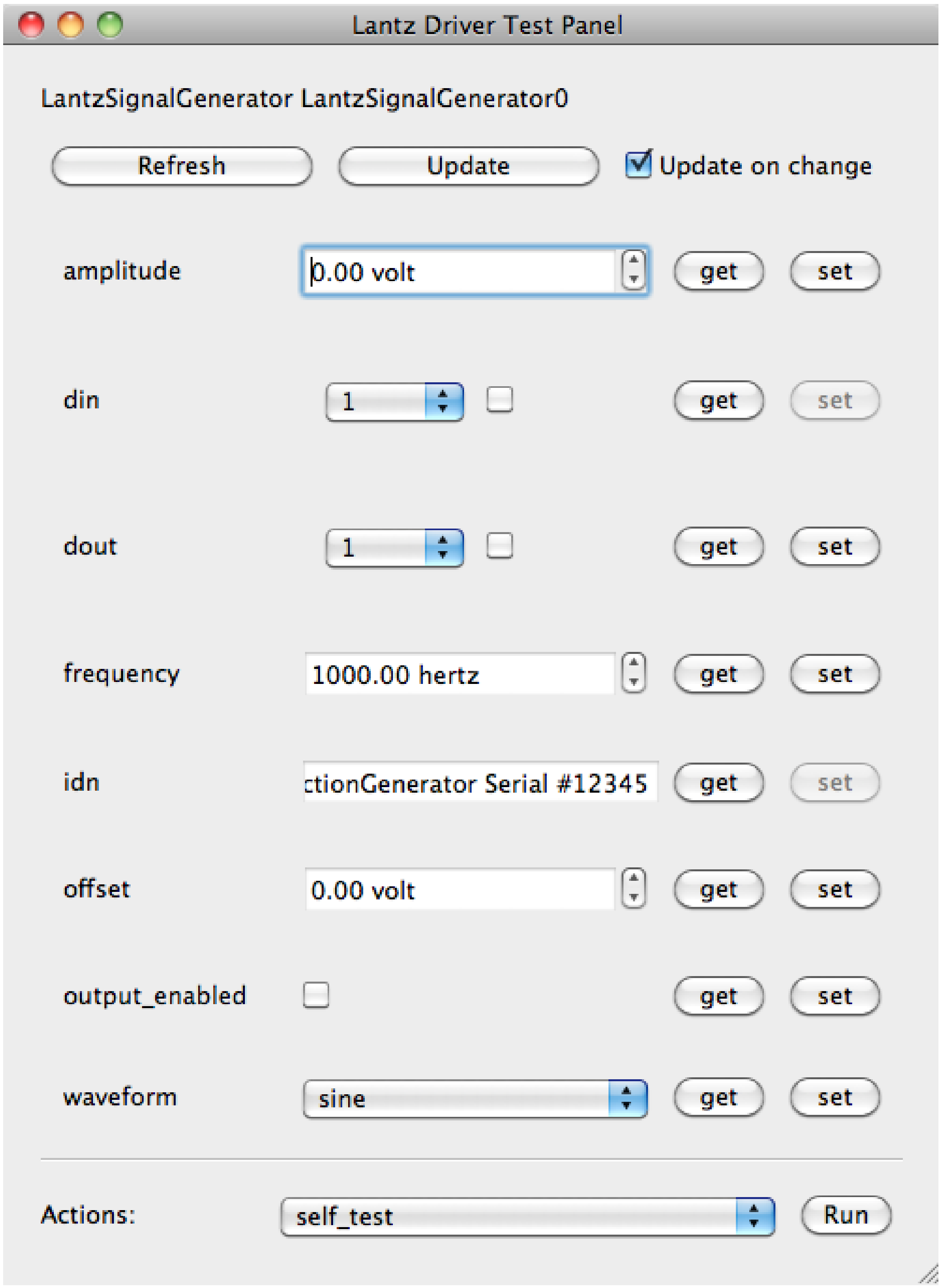
November 20, 2016

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.

When you use Lantz you get:

- A comprehensive and growing library of curated and well documented instruments drivers.
- A really easy way write your own drivers. In less than an hour you can write a full driver with bounds checks, useful logging, async capabilities and much more.
- On-the-fly GUI for testing purposes. Without a line of code you get for **any** driver something like this (click to enlarge):



LantzSignalGenerator LantzSignalGenerator0

[✓] Update on change

amplitude: 0.00 volt

din: 1 ☐

dout: 1 ☐

frequency: 1000.00 hertz

idn: ctionGenerator Serial #12345

offset: 0.00 volt

output_enabled: ☐

waveform: sine

Actions: self_test

- Tools to quickly build beautiful, responsive and composable applications.
- Rapid application development primitives.

- An awesome and supporting community.

CHAPTER 1

More information

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 2

Learn

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

Guides
Task oriented guides.

More information

Contributing
We need your brain.

Drivers
List of Lantz drivers.

API
Application programming interface reference.

CHAPTER 4

Indices and tables

genindex
Lists all sections and subsections.

modindex
All functions, classes, terms.

search
Search this documentation.

4.1 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 its 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

4.2 Overview

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

```
from lantz import Q_

from lantz.drivers.aeroflex import A2023a

fungen = A2023a('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!

4.2.1 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.A2023a.A2023a0
```

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

```
fungen = A2023a('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!

4.2.2 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')
```

4.2.3 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 }
```

4.2.4 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.

4.2.5 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

>>> 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.

4.2.6 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

4.2.7 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 A2023a('COM1') as fungen:

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

4.2.8 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 is 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.amplitude = 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.amplitude = 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 A2023a
from lantz.drivers.standford import SR844

with FrequenceMeter('COM1') as fmeter, \
     A2023a('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

4.2.9 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:

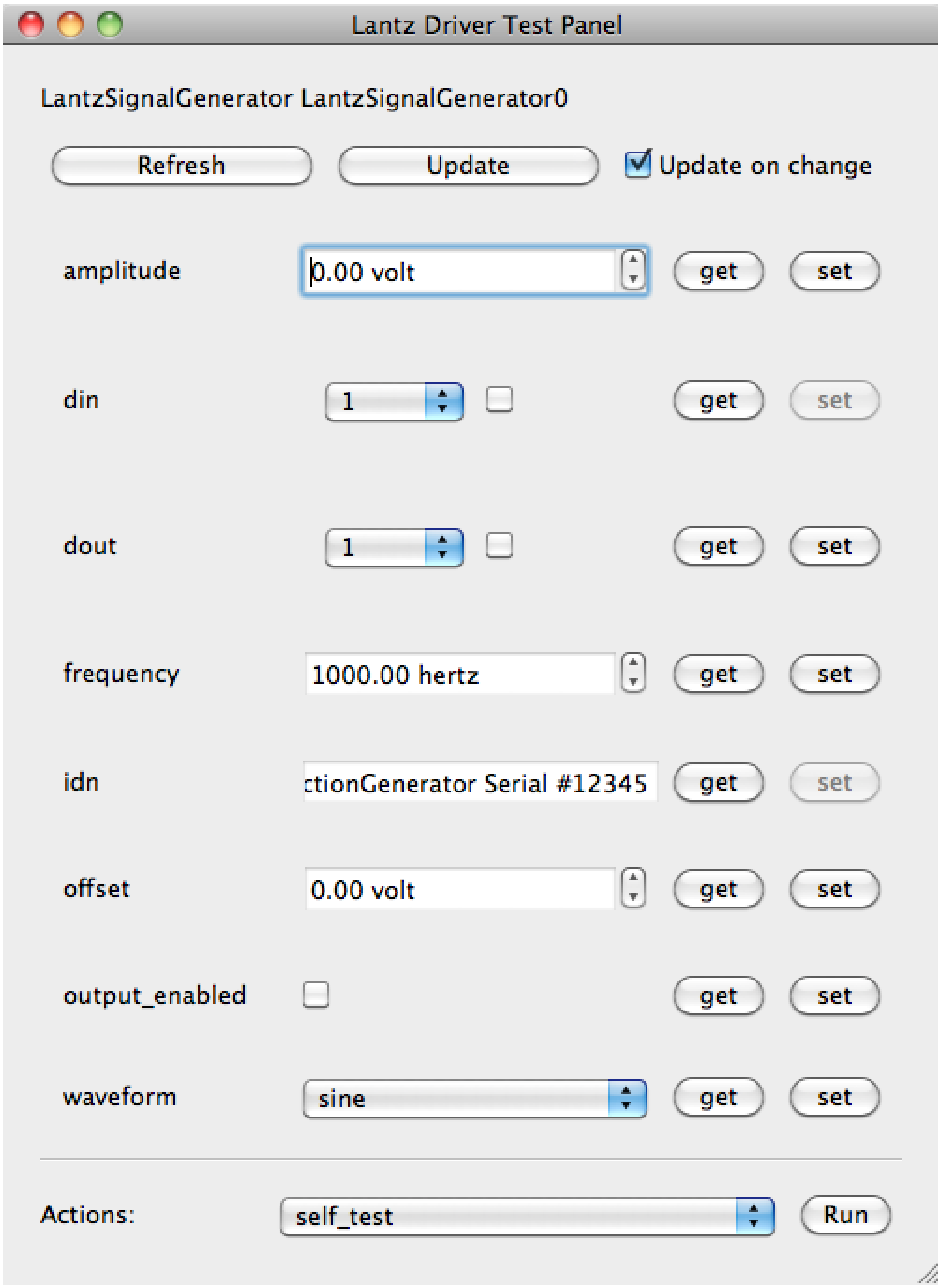
```
# imports not shown

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:



LantzSignalGenerator LantzSignalGenerator0

[✓] 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

Check out the *Tutorials* to get started!

4.3 Tutorials

4.3.1 Installation guide

This guide describes Lantz requirements and provides platform specific installation guides. Examples are given for Python 3.4 installing all optional requirements as site-packages.

Requirements

Lantz core requires Python 3.4+ and:

- PyVISA Python package that enables you to control all kinds of measurement devices independently of the interface (e.g. GPIB, RS232, USB, Ethernet) using different backends.
- Qt4 is used to generate the graphical user interfaces. Due to a license issue there are two python bindings for Qt: PyQt and PySide.

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.
- 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*

Note: A really simple way that works in all OS is using Anaconda Python Distribution (see below)

Linux

Most linux distributions provide packages for Python 3.4, 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 3.4
2. (optionally) Install PyQt, NumPy
3. (optionally) Install VISA
4. Open a terminal to install pip:

```
    $ curl http://python-distribute.org/distribute_setup.py | python3.4
    $ curl https://raw.github.com/pypa/pip/master/contrib/get-pip.py | python3.4
```

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

```
    $ pip-3.4 install sphinx pyserial colorama lantz
```

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 3.4+.

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 3.4:

```
- 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:\Python3.4\Scripts\pip install sphinx pyserial colorama lantz
```

Anaconda

Anaconda is a Scientific Oriented Python Distribution. It can be installed in Linux, OSX and Windows; without administrator privileges. It has a binary package manager that makes it really easy to install all the packages that you need to use Lantz (and much more!)

It comes in two flavors: miniconda and anaconda, which is is just miniconda with a lot of predefine packages. Here we show you how to do it with miniconda.

In any OS you can use Anaconda Python Distribution

1. Download and install the apropriate miniconda3 for your OS. The easiest way is that you download miniconda3 to get Python 3 as default Both 32 and 64 bits are ok

   Warning: Make sure that all subsequents command are executed using the miniconda binaries.

2. If you want a minimal environment:

   ```
   $ conda install pip numpy sphinx pyqt
   ```

   or if you want everything:

   ```
   $ conda install anaconda
   ```

4. Install Lantz:

   ```
   $ pip install colorama pyserial pyusb lantz
   ```

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.

4.3.2 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:\Python34\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:

```python
from lantz.drivers.examples import LantzSignalGenerator

inst = LantzSignalGenerator('TCPIP::localhost::5678::SOCKET')
inst.initialize()
print(inst.idn)
inst.finalize()
```

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

```python
from lantz.drivers.examples import LantzSignalGenerator
```

Instead of memorizing lot of text based commands and fighting with formatting and parsing, Lantz provides an **object oriented layer** to instrumentation. In this case, the driver for our simulated device is under the company *examples* and is named *LantzSignalGenerator*.

Then we create an instance of the class:

```python
inst = LantzSignalGenerator('TCPIP::localhost::5678::SOCKET')
```

The string 'TCPIP::localhost::5678::SOCKET' is called the **Resource Name** and is specified by the VISA specification. It specifies the how you connect to your device. Lantz uses the power of VISA through PyVISA where you can also find a description of the VISA resource names.

Notice that is the resource name, not the driver what specifies the connectivity. This allows easy programming of instruments supporting multiple protocols.

We then connect to the device by calling the `initialize()` method:

```python
inst.initialize()
```

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

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

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

```python
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
```

(You can also run it in the python console)

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 *lantz-sim* 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. We will see more about this later on when we start *Building your own drivers*

Safely releasing resources

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

```
from lantz.drivers.examples import LantzSignalGenerator

inst = LantzSignalGenerator('TCPIP::localhost::5678::SOCKET')
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 LantzSignalGenerator

inst = LantzSignalGenerator('TCPIP::localhost::5678::SOCKET')
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 LantzSignalGenerator

with LantzSignalGenerator('TCPIP::localhost::5678::SOCKET') 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 LantzSignalGenerator

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

with LantzSignalGenerator('TCPIP::localhost::5678::SOCKET') as inst:
    print(inst.idn)
    print(inst.waveform)
```

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

```
14:38:43 INFO     Created LantzSignalGenerator0
14:38:43 DEBUG    Using MessageBasedDriver for TCPIP::localhost::5678::SOCKET
14:38:43 INFO     Calling initialize
14:38:43 INFO     initialize returned None
14:38:43 DEBUG    Opening resource TCPIP::localhost::5678::SOCKET
14:38:43 DEBUG    Setting [('write_termination', '\n'), ('read_termination', '\n')]
14:38:43 INFO     Getting idn
14:38:43 DEBUG    Writing '?IDN'
14:38:43 DEBUG    Read 'FunctionGenerator Serial #12345'
14:38:43 DEBUG    (raw) Got FunctionGenerator Serial #12345 for idn
14:38:43 INFO     Got FunctionGenerator Serial #12345 for idn
14:38:43 INFO     Getting waveform
14:38:43 DEBUG    Writing '?WVF'
14:38:43 DEBUG    Read '0'
14:38:43 DEBUG    (raw) Got 0 for waveform
14:38:43 INFO     Got sine for waveform
14:38:43 DEBUG    Closing resource TCPIP::localhost::5678::SOCKET
14:38:43 INFO     Calling finalize
14:38:43 INFO     finalize returned None
FunctionGenerator Serial #12345
sine
```

The first line shows the creation of the driver instance. As no name was provided, Lantz assigns one (*LantzSignal-Generator0*). 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.

You can change the name of the instrument when you instantiate it. Also change *DEBUG* to *INFO* run it again to see the different levels of information you can get.

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

from lantz.drivers.examples import LantzSignalGenerator

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

with LantzSignalGenerator('TCPIP::localhost::5678::SOCKET', name='my-device') 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 LantzSignalGenerator

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

with LantzSignalGenerator('TCPIP::localhost::5678::SOCKET') as inst:
    print(inst.idn)
    print(inst.idn)
    print(inst.waveform)
    print(inst.waveform)
```

If you see the log output:

```
14:42:32 INFO     Created LantzSignalGenerator0
14:42:32 DEBUG    Using MessageBasedDriver for TCPIP::localhost::5678::SOCKET
14:42:32 INFO     Calling initialize
14:42:32 INFO     initialize returned None
14:42:32 DEBUG    Opening resource TCPIP::localhost::5678::SOCKET
14:42:32 DEBUG    Setting [('read_termination', '\n'), ('write_termination', '\n')]
14:42:32 INFO     Getting idn
14:42:32 DEBUG    Writing '?IDN'
14:42:32 DEBUG    Read 'FunctionGenerator Serial #12345'
14:42:32 DEBUG    (raw) Got FunctionGenerator Serial #12345 for idn
14:42:32 INFO     Got FunctionGenerator Serial #12345 for idn
14:42:32 INFO     Getting waveform
14:42:32 DEBUG    Writing '?WVF'
14:42:32 DEBUG    Read '0'
14:42:32 DEBUG    (raw) Got 0 for waveform
14:42:32 INFO     Got sine for waveform
14:42:32 INFO     Getting waveform
14:42:32 DEBUG    Writing '?WVF'
14:42:32 DEBUG    Read '0'
14:42:32 DEBUG    (raw) Got 0 for waveform
14:42:32 INFO     Got sine for waveform
14:42:32 DEBUG    Closing resource TCPIP::localhost::5678::SOCKET
14:42:32 INFO     Calling finalize
14:42:32 INFO     finalize returned None
FunctionGenerator Serial #12345
FunctionGenerator Serial #12345
sine
```

```
sine
```

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 LantzSignalGenerator

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

with LantzSignalGenerator('TCPIP::localhost::5678::SOCKET') as inst:
    inst.waveform = 'sine'
    inst.waveform = 'sine'
```

the log output:

```
14:44:09 INFO     Created LantzSignalGenerator0
14:44:09 DEBUG    Using MessageBasedDriver for TCPIP::localhost::5678::SOCKET
14:44:09 INFO     Calling initialize
14:44:09 INFO     initialize returned None
14:44:09 DEBUG    Opening resource TCPIP::localhost::5678::SOCKET
14:44:09 DEBUG    Setting [('write_termination', '\n'), ('read_termination', '\n')]
14:44:09 INFO     Setting waveform = sine (current=MISSING, force=False)
14:44:09 DEBUG    (raw) Setting waveform = 0
14:44:09 DEBUG    Writing '!WVF 0'
14:44:09 DEBUG    Read 'OK'
14:44:09 INFO     waveform was set to sine
14:44:09 INFO     No need to set waveform = sine (current=sine, force=False)
14:44:09 DEBUG    Closing resource TCPIP::localhost::5678::SOCKET
14:44:09 INFO     Calling finalize
14:44:09 INFO     finalize returned None
```

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 LantzSignalGenerator

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

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

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

```
14:44:44 INFO     Created LantzSignalGenerator0
14:44:44 DEBUG    Using MessageBasedDriver for TCPIP::localhost::5678::SOCKET
14:44:44 INFO     Calling initialize
14:44:44 INFO     initialize returned None
14:44:44 DEBUG    Opening resource TCPIP::localhost::5678::SOCKET
14:44:44 DEBUG    Setting [('read_termination', '\n'), ('write_termination', '\n')]
```

```
14:44:44 INFO     Setting waveform = sine (current=MISSING, force=False)
14:44:44 DEBUG    (raw) Setting waveform = 0
14:44:44 DEBUG    Writing '!WVF 0'
14:44:44 DEBUG    Read 'OK'
14:44:44 INFO     waveform was set to sine
14:44:44 INFO     Setting waveform = sine (current=sine, force=True)
14:44:44 DEBUG    (raw) Setting waveform = 0
14:44:44 DEBUG    Writing '!WVF 0'
14:44:44 DEBUG    Read 'OK'
14:44:44 INFO     waveform was set to sine
14:44:44 DEBUG    Closing resource TCPIP::localhost::5678::SOCKET
14:44:44 INFO     Calling finalize
14:44:44 INFO     finalize returned None
```

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 LantzSignalGenerator

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

with LantzSignalGenerator('TCPIP::localhost::5678::SOCKET') 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.

4.3.3 Using Feats

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

```
from lantz.drivers.examples import LantzSignalGenerator

with LantzSignalGenerator('TCPIP::localhost::5678::SOCKET') 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 LantzSignalGenerator

with LantzSignalGenerator('TCPIP::localhost::5678::SOCKET') 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.LantzSignalGenerator), *output_enabled* accepts only *True* or *False*. If your provide a different value:

```
from lantz.drivers.examples import LantzSignalGenerator

with LantzSignalGenerator('TCPIP::localhost::5678::SOCKET') 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 LantzSignalGenerator

with LantzSignalGenerator('TCPIP::localhost::5678::SOCKET') 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 LantzSignalGenerator
from lantz import Q_

with LantzSignalGenerator('TCPIP::localhost::5678::SOCKET') 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 LantzSignalGenerator
from lantz import Q_

with LantzSignalGenerator('TCPIP::localhost::5678::SOCKET') 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.

4.3.4 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 (
it is yours) and change it to look like this:

```python
from lantz import Feat
from lantz.messagebased import MessageBasedDriver

class LantzSignalGenerator(MessageBasedDriver):
    """Lantz Signal Generator.
    """

    DEFAULTS = {'COMMON': {'write_termination': '\n',
                           'read_termination': '\n'}}

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

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

The code is straight forward. We first import :class:MessageBasedDriver from :mod:lantz.messagebased (the Lantz module for message based instruments). MessageBasedDriver is a base class (derived from :class:Driver) that implements methods to communicate via different protocols. 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 method present in :class:MessageBasedDrivers. *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 :ref:_defaults_dictionary at the beginning of the class. You can find more information about in the guides, but for now on, we will just point out that the key **COMMON** is used to tell Lantz that the following keyword arguments are for all instrument types (USB, GPIB, etc). In particular we specify that the read and write termination are *'n'*.

Finally, inside the __*name*__ == '__*main*__' we instantiate the SignalGenerator (as we have seen in :ref:using 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* in the console.

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

```
from lantz import Feat
from lantz.network import MessageBasedDriver

class LantzSignalGenerator(MessageBasedDriver):
    """Lantz Signal Generator.
    """

    DEFAULTS = {'COMMON': {'write_termination': '\n',
                           'read_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 LantzSignalGenerator('TCPIP::localhost::5678::SOCKET') 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 *lantz-sim*. 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 LantzSignalGenerator(MessageBasedDriver):

    # 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 LantzSignalGenerator(MessageBasedDriver):

    # 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 MessageBasedDriver
from lantz.errors import InstrumentError

class LantzSignalGenerator(MessageBasedDriver):
    """Lantz Signal Generator.
    """

    DEFAULTS = {'COMMON': {'write_termination': '\n',
                           'read_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 LantzSignalGenerator('TCPIP::localhost::5678::SOCKET') 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 MessageBasedDriver
from lantz.errors import InstrumentError

class LantzSignalGenerator(MessageBasedDriver):
    """Lantz Signal Generator
    """

    DEFAULTS = {'COMMON': {'write_termination': '\n',
                           'read_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 MessageBasedDriver
from lantz.errors import InstrumentError

class LantzSignalGenerator(MessageBasedDriver):
    """Lantz Signal Generator
    """

    DEFAULTS = {'COMMON': {'write_termination': '\n',
                           'read_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 LantzSignalGenerator('TCPIP::localhost::5678::SOCKET') 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.

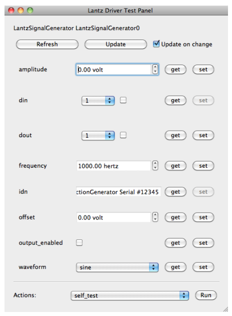
This means that we can write the body of the getter/setter expecting a instrument compatible value (1 or 0) but the user actually sees a much more friendly interface (True or False). The same happens with *waveform*. Instead of asking the user to memorize which number corresponds to 'sine' or implement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

Beautiful Testing

Testing in the command line is extermely useful but sometimes it is desirable to have simple GUI to be used as a test panel. Lantz gives you that with no effort. Just change the main part to this:

```
if __name__ == '__main__':
    from lantz.ui.qtwidgets import start_test_app

    with LantzSignalGenerator('TCPIP::localhost::5678::SOCKET') as inst:
        start_test_app(inst)
```

and you will get something like this:



LantzSignalGenerator LantzSignalGenerator0

☑ Update on change

```
amplitude       0.00 volt
din             1  ☐
dout            1  ☐
frequency       1000.00 hertz
idn             ctionGenerator Serial #12345
offset          0.00 volt
output_enabled  ☐
waveform        sine
Actions:        self_test
```

Cool, right? Using Python amazing introspection capabilites together with Feat annotations inside the driver, Lantz was able to build **on-the-fly** a Graphical User Interface for testing purposes.

Among other things, you get:

- The right widget, for the right datatype: Check-box for boolean, Combo-box for options, etc.
- When a Feat has units, the suffix is displayed. You can also press the *u* key to change the displayed units.
- Minimum and Maximum values when a Feat has limits

... and much more! All of this without an extra line of code.

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 LantzSignalGenerator(MessageBasedDriver):

    # 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 LantzSignalGenerator(MessageBasedDriver):

    # 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 LantzSignalGenerator(MessageBasedDriver):

    # 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.

4.3.5 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 LantzSignalGenerator

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

with LantzSignalGenerator('TCPIP::localhost::5678::SOCKET') 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 LantzSignalGenerator

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 LantzSignalGenerator('TCPIP::localhost::5678::SOCKET') 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
```

Learn how in the next part of the tutorial: A simple GUI app.

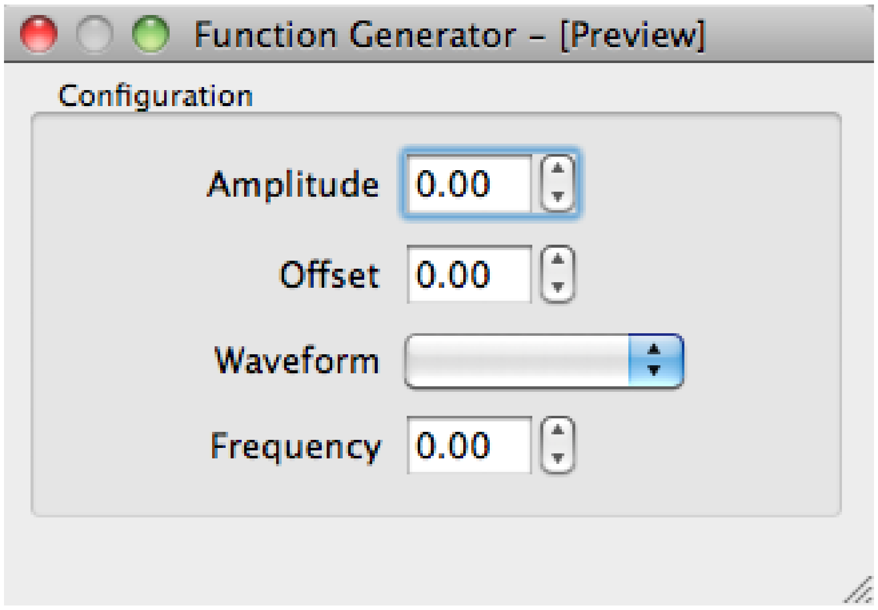
4.3.6 A simple GUI app

In this part of the tutorial you will build an application to custom function generator GUI:

Start the simulated instrument running the following command:

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

Using Qt Designer, create a window like this:



and save it as *fungen.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). 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 *fungen-gui.py* with the following content:

```
import sys

# Import from a lantz the start_gui helper function
from lantz.ui.app import start_gui

#
with LantzSignalGenerator('TCPIP::localhost::5678::SOCKET') as inst:
    start_gui('fungen.ui', inst)
```

Run it and enjoy:

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

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

:func:start_gui take at leas two arguments. First, the fullpath of an QtDesigner ui file. As a second argument, an instrument instance.

Under the hood, *start_gui* is creating a Qt Application and loading the ui file. Then it 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 fine grained alternatives in *Connecting a custom UI to a driver*.

Learn how in the next part of the tutorial: A simple GUI app.

4.3.7 A rich GUI app

Building a rich, responsive and reusable app is tough. In this part of the tutorial you learn how Lantz makes it simpler by build an application using Blocks and :class:Backend and :class:Frontend classes.

This is the wish list for our application:

- Should measure with a voltmeter as we scan the frequency in a function generator.
- The user should be able to change the range and step of the frequency scan.
- The user should be able to cancel the scan at any moment.
- Plot the results live

First, as we have done before start the simulated function generator running the following command:

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

We will also start simulated instrument running the following command in another terminal:

```
$ lantz-sim voltmeter tcp -p 5679
```

Let's test our two instruments before start:

```python
from lantz import Q_
from lantz.drivers.examples import LantzSignalGenerator, LantzVoltmeter

# We change the frequency of the function generator
with LantzSignalGenerator('TCPIP::localhost::5678::SOCKET') as fungen:
    print(fungen.idn)
    fungen.frequency = Q_(3.14, 'Hz')

# We measure the voltage in channel 0 of the voltmeter
with LantzVoltmeter('TCPIP::localhost::5679::SOCKET') as voltmeter:
    print(voltmeter.idn)
    print(voltmeter.voltage[0])
```

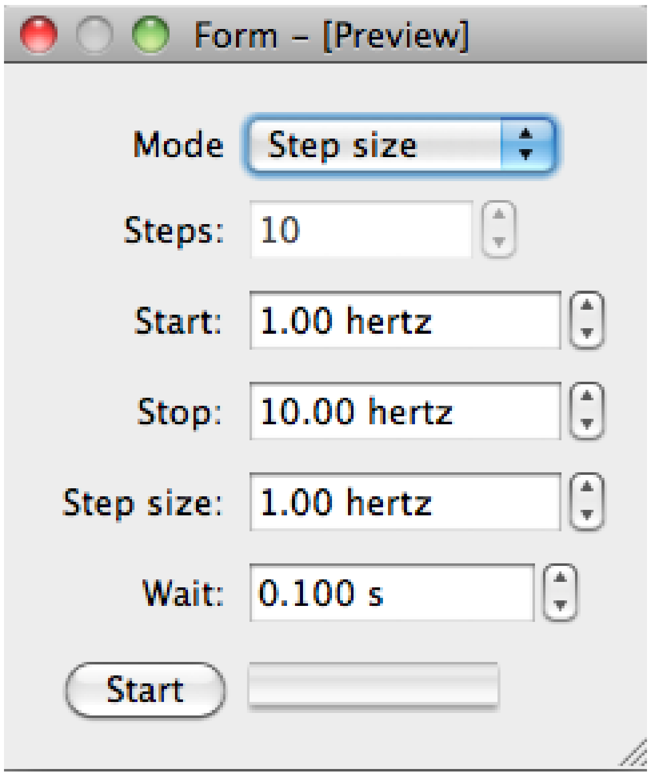
Now that everything works, let's make the app!

The scanner

Scanning (a frequency, a voltage, etc.) is a very common task in instrumentation applications. That is why lantz provide a building block (usually called just blocks) for this. It is called :class:FeatScan

```python
# We import a helper function to start the app
from lantz.ui.app import start_gui_app

# The block consists of two parts the backend and the frontend
from lantz.ui.blocks import FeatScan, FeatScanUi

# An this you know already
from lantz.drivers.examples import LantzSignalGenerator

with LantzSignalGenerator('TCPIP::localhost::5678::SOCKET') as fungen:

    # Here we instantiate the backend setting 'frequency' as the Feat to scan
    # and specifying in which instrument
    app = FeatScan('frequency', instrument=fungen)

    # Now we use the helper to start the app.
    # It takes a Backend instance and a FrontEnd class
    start_gui_app(app, FeatScanUi)
```

Save it, run it and how the Feat is scanned in the simulator. In no time you build a responsive and interactive application in which you can choose the start and stop value, the number of steps or the step size, the interval between calls like the one you see below:



Mode: Step size
Steps: 10
Start: 1.00 hertz
Stop: 10.00 hertz
Step size: 1.00 hertz
Wait: 0.100 s

Blocks are small, reusable, composable and easy to use. They do one thing, and they do it right. Combining multiple blocks you can build a complex and rich application. There are many available blocks and you can build your own. We will not go into details right now but it is only important that you know that blocks can derive from one of two classes: :class:Backend and :class:Frontend. The first contains the logic of your application and the second the GUI. This keeps things easier to test and develop. It also allows you to call you application without any GUI (for example from the command line) or with a different GUI (for example a debug GUI with more information).

Measuring

Now we need to measure in the voltmeter in each step. There is a really simple way to do it. A :class:FeatScan object exposes an attribute (*body*) in which you can hook a function. The function should take three arguments:

- counter: the step number from 0..N-1
- new_value: the feat value that was used
- overrun: a boolean indicating that executing the body is taking longer than the interval that you have allocated.

```
# We import a helper function to start the app
from lantz.ui.app import start_gui_app

# The block consists of two parts the backend and the frontend
from lantz.ui.blocks import FeatScan, FeatScanUi

# An this you know already
from lantz.drivers.examples import LantzSignalGenerator, LantzVoltmeter

with LantzSignalGenerator('TCPIP::localhost::5678::SOCKET') as fungen, \
     LantzVoltmeter('TCPIP::localhost::5679::SOCKET') as voltmeter:

    def measure(counter, new_value, overrun):
        print(new_value, voltmeter.voltage[0])

    app = FeatScan('frequency', instrument=fungen)

    # Here we are telling the FeatScan backend to call the measure function
    # in each scan step. It will build a
    app.body = measure

    # Now we use the helper to start the app.
    # It takes a Backend instance and a FrontEnd class
    start_gui_app(app, FeatScanUi)
```

That's it! You can put in the body anything that you like: waiting, changing the scale, etc. The only rule is that the backend is not aware of the Frontend. So ... how are we going to plot?

Composing an application

We are going to create an application embedding the FeatScan. In the backend we will add an InstrumentSlot for the voltmeter: this tells Lantz that an instrument is necessary. We will also add a Signal to tell the GUI to plot new data.

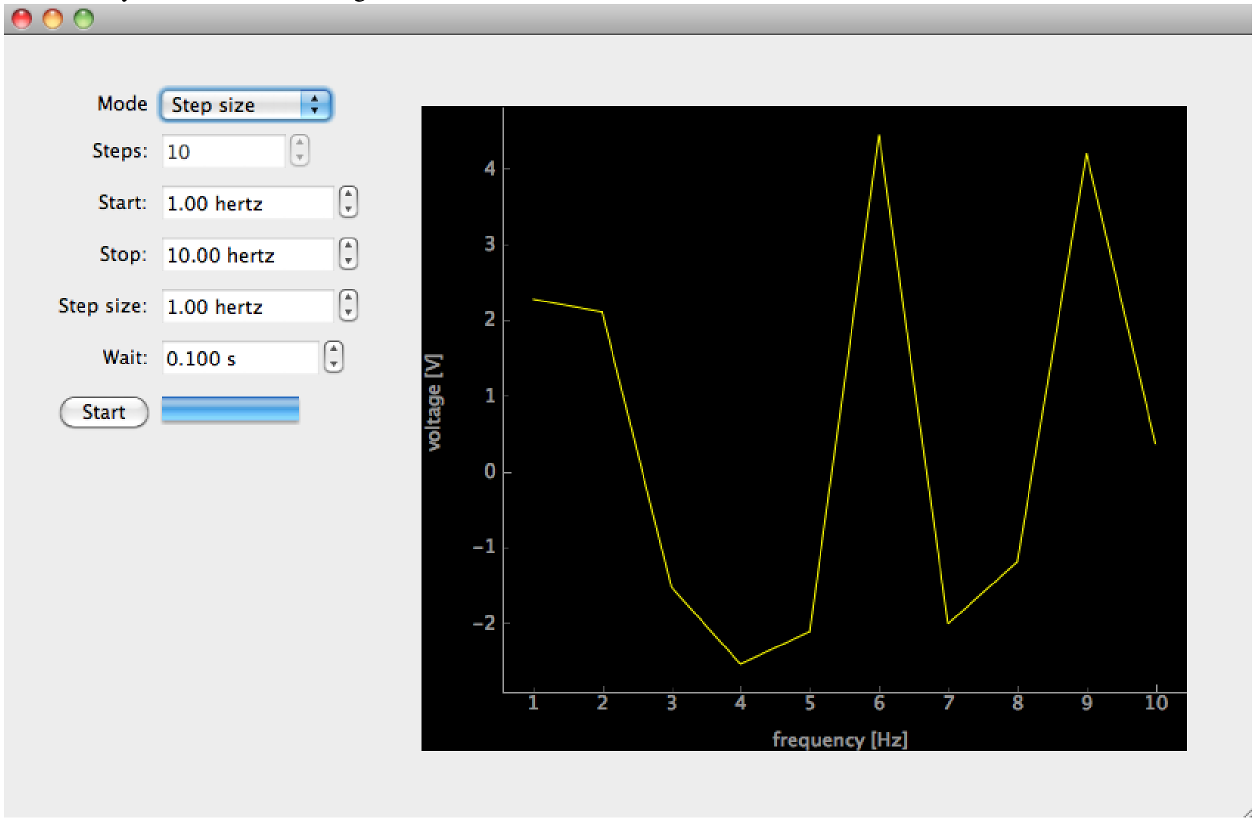
Note: Why a signal? Qt Signals is a way of async communication between objects. It is a way in which one object can inform others that something has happend. A signal is emitted by an object and received by another in an slot (a function). You need connect signals and slots for this to happen. Why we cannot just call the frontend? If you call the frontend, the backend will not be able to do anything until the frontend finishes. Emitting a signal tells the frontend to do something but without disturbing the backend job.

In the frontend, we will connect this signal to a Plot function (you will need pyqtgraph for this). Try installing it with:

```
$ pip install pyqtgraph
```

```
# We import a helper function to start the app
from lantz.ui.app import start_gui_app

# Import Qt modules from lantz (pyside and pyqt compatible)
from lantz.utils.qt import QtCore

# We import the FeatScan backend and
# the ChartUi a frontend with a chart.
# You require pyqtgraph for this.
from lantz.ui.blocks import FeatScan, FeatScanUi, ChartUi, HorizonalUi

from lantz.ui.app import Backend, start_gui_app, InstrumentSlot

# We first create our backend
class MyApp(Backend):

    # We embed the FeatScan app.
    # Notice that we put the class, not an instance of it.
    scanner = FeatScan

    # The app needs an instrument that will be available in the voltmeter attribute
    # It also needs another instrument to scan, but this is included in FeatScan
    voltmeter = InstrumentSlot

    # This signal will be emitted when new data is available.
    # The two values are the x and y values
    new_data = QtCore.Signal(object, object)


    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)

        # We assign the scanner body to our function.
        self.scanner.body = self.measure

    def measure(self, counter, new_value, overrun):

        # We measure and we emit a signal.
        # Remember that these values have units!!
        self.new_data.emit(new_value, self.voltmeter.voltage[0])

# This will be our Frontend
# We inherite from HorizonalUi, which organizes the widgets automatically horizontally
class MyGui(HorizonalUi):

    # We embed two existing Frontends. Notice that agian

    # The FeatScanUi, which you know alredy.
    # But we say that it will be using the scanner backend
    # Notice that we put the class, not an instance of it.
    scanui = FeatScanUi.using('scanner')

    # The ChartUi, which plot a dataset point-by-point using pyqtgraph
    chartui = ChartUi

    # Here we tell HorizonalUi how we want to organize the widgets
    # Notice that we need to put the names of the attributes as strings.

    parts = ('scanui',     # The FeatScanUi will be in the first colum
                           #  and connected to the embedded scanner backend
             'chartui')    # The ChartUI will be in the second column.

    def connect_backend(self):

        # This method is called after gui has been loaded (referenced in self.widget)
        # and the backend is connected to the frontend (referenced in self.backend).
        # In this case, we use it to connect the new_data signal of the backend
        # with the plot function in ChartUi
        self.backend.new_data.connect(self.chartui.plot)

        # To clear the chart every time we start a new scan
        # we connect the request start signal of the user interface
        # to the clear method of the chart ui
        self.scanui.request_start.connect(self.chartui.clear)

        super().connect_backend()

        # We define the labels and the units to use

        # For the y axis, it is fixed.
        self.chartui.ylabel = 'voltage'
        self.chartui.yunits = 'V'
        # For the x axis, depends on the feat selected by the user.
        self.chartui.xlabel = self.backend.scanner.feat_name
        self.chartui.xunits = self.backend.scanner.feat_units

        # Notice that the units is not just a change to the label,
        # it rescales the values that are shown in the plot.


if __name__ == '__main__':
    # An this you know already
    from lantz.drivers.examples import LantzSignalGenerator, LantzVoltmeter

    with LantzSignalGenerator('TCPIP::localhost::5678::SOCKET') as fungen, \
         LantzVoltmeter('TCPIP::localhost::5679::SOCKET') as voltmeter:

        app = MyApp(instrument=fungen, voltmeter=voltmeter,
                    scanner={'feat_name': 'frequency'})

        # Now we use the helper to start the app.
        # It takes a Backend instance and a FrontEnd class
        start_gui_app(app, MyGui)
```

Run it and you will see something like this:


Mode: Step size
Steps: 10
Start: 1.00 hertz
Stop: 10.00 hertz
Step size: 1.00 hertz
Wait: 0.100 s

There is much more to know. Hopefully this tutorial get's you started.

4.4 Guides

This are tasks oriented guides:

4.4.1 Using drivers

Avoiding Resource Names: the *via* methods

While resource names provide a comprehensive way to address devices in many cases you want a simpler and succint way to do it. :class::MessageBasedDriver provide special methods for each interface type. Under the hood, these methods build the resource name for you based on minimum information.

Via Serial

For serial instruments, you just need to provide the serial port. Instead of doing:

```
with MyDriver('ASRL1::INSTR') as instrument:

    print(instrument.idn)
```

you can do:

```
with MyDriver.via_serial(1) as instrument:

    print(instrument.idn)
```

Just like with the standard constructor you can specify the name of the instrument (for logging purposes):

```
with MyDriver.via_serial(1, name='mydevice') as instrument:

    print(instrument.idn)
```

And you can also specify the initialization settings:

```
with MyDriver.via_serial(1, name='mydevice', 'read_termination'='\n') as instrument:

    print(instrument.idn)
```

Names and setting are available for all constructor methods so we will ignore them from now on.

Via TCPIP

For TCPIP instruments, you just need to provide the hostname (ip address) and port. Instead of doing:

```
with MyDriver('TCPIP::localhost::5678::INSTR') as instrument:

    print(instrument.idn)
```

you can do:

```
with MyDriver.via_tcpip('localhost', 5678) as instrument:

    print(instrument.idn)
```

The same is true for TCIP Sockets.

Instead of doing:

```
with MyDriver('TCPIP::localhost::5678::SOCKET') as instrument:

    print(instrument.idn)
```

you can do:

```
with MyDriver.via_tcpip_socket('localhost', 5678) as instrument:

    print(instrument.idn)
```

Via GPIB

For TCPIP instruments, you just need to provide the gpib address:

```
with MyDriver('GPIB::9::INSTR') as instrument:

    print(instrument.idn)
```

you can do:

```
with MyDriver.via_gpib(9) as instrument:

    print(instrument.idn)
```

Via USB

For USB instruments the thing becomes really useful. USB devices *announce* themselves to the computer indicating the manufacturer id, model code and serial number. Many classes have hardcoded the manufacturer id and model code (or codes) of the instruments they can control.

So instead of doing:

```
with MyDriver('USB0::0x1AB1::0x0588::DS1K00005888::INSTR') as instrument

    print(instrument.idn)
```

you can just do:

```
with MyDriver.via_usb() as instrument:

    print(instrument.idn)
```

If you have multiple identical usb instuments connected, this will fail. In this case you need to specify the serial number of the instrument you want:

```
with MyDriver.via_usb('DS1K00005888') as instrument:

    print(instrument.idn)
```

If you want to try if a driver can control another instrument you can override the model code and/or the manufacturer id:

```
with MyDriver.via_usb(manufacturer_id='0x1AB2', model_code='0x0589') as instrument:

    print(instrument.idn)
```

You can also specify the serial code if you want. The rule is simple you need to specify it in a way that there is only one.

The same arguments are valid to create a USB SOCKET:

```
with MyDriver.via_usb_socket() as instrument:

    print(instrument.idn)
```

Initializing multiple drivers

Initializing and finalizing drivers correctly is an important aspect of any instrumentation application. In particular, if resources are left open after the program finishes it can lead to deadlocks.

Lantz provides context managers to ensure that that these methods are called. For example:

```
with A2023a.via_serial(1) as fungen:

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

will call the initializer in the first line and the finalizer when the program exits the block even in the case of an unhandled exception as explained in *Safely releasing resources*.

This approach is very useful but inconvenient if the number of instruments is large. For three instruments is still fine:

```
with FrequenceMeter.via_serial(1) as fmeter, \
     A2023a.from_serial_port(2) as fungen, \
     SR844.from_serial_port(3) as lockin:

    freq = fmeter.frequency

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

but for a larger number it will be annoying. In addition in GUI applications, you might want to initialize the drivers when a button is pressed so wrapping the code in a *with* statement is not an option.

Lantz provides *initialize_many* and *finalize_many* to solve this problem.

The previous example will look like this:

```
drivers = (FrequenceMeter.via_serial(1), A2023a.via_serial(2), SR844.via_serial(3))

initialize_many(drivers)
```

Under the hood, *initialize_many* is calling the *initialize* method of each driver and registering the *finalize* method to be executed when the Python interpreter exits.

If you want to call the finalizers manually, you can:

```
initialize_many(drivers, register_finalizer=False)
# Do Something
finalize_many(drivers)
```

Providing status

You can define two callback functions to be executed before and after driver initialization:

```
start = time.time()
def initializing(driver):
    print('Initializing ' driver.name)

def initialized(driver):
        print('Initialized {} in {} secs'.format(driver.name, time.time() - start))

initialize_many(drivers, on_initializing=initializing, on_initialized=initialized)
print('Done in {:.1f} secs'.format(time.time() - start))
```

will print (if we assume for each instrument certain initialization times):

```
Initializing FrequenceMeter1
Initialized FrequenceMeter1 in 3.2 secs
Initializing A2023a1
Initialized A2023a1 in 4.1 secs
Initializing SR8441
Initialized SR8441 in 3.5 secs
Done in 10.8 seconds.
```

Initializing and *Initialized* are interleaved as the initialization of all drivers occurs serially.

Similarly, in *finalize_many* you can use *on_finalizing* and *on_finalized*.

Faster initialization and finalization

In most cases, the initialization of each driver is independent of the rest and a significant speed up can be achieved by running the tasks concurrently:

```
initialize_many(drivers, on_initializing=initializing, on_initialized=initialized, concurrent=True)
```

The output will no be:

```
Initializing FrequenceMeter1
Initializing SR8441
Initializing A2023a1
Initialized FrequenceMeter1 in 3.2 secs
Initialized SR8441 in 3.5 secs
Initialized A2023a1 in 4.1 secs
Done in 4.1 seconds.
```

Initialization is now done concurrently yielding 2x speed up. For a larger number of instruments, the speed up will be even larger.

You can also use this argument in *finalize_many*.

Initialization hierarchy

If a particular order in the initialization is required, you can order the list (or tuple) and do a serial (concurrent=False) initialization. But is slow again.

You can specify a hierarchy of initialization using the *dependencies* argument. If the A2023a1 requires that SR8441 and FrequenceMeter1 are initialized before, the call will be:

```
initialize_many(drivers, on_initializing=initializing, on_initialized=initialized,
                concurrent=True, dependencies={'A2023a1': ('SR8441', 'FrequenceMeter1')})
```

and the result will be:

```
Initializing FrequenceMeter1
Initializing SR8441
Initialized FrequenceMeter1 in 3.2 secs
Initialized SR8441 in 3.5 secs
Initializing A2023a1
Initialized A2023a1 in 4.1 secs
Done in 7.6 seconds.
```

The *dependencies* argument takes a dictionary where each key is a driver name and the corresponding value is a list of the drivers names that need to be initialized before. It can have arbitrary complexity. If a driver is not present in the dictionary, it will be initialized with the ones without dependencies.

You can use these arguments also in *finalize_many*, but the requirements are interpreted in reverse. This allows to use the same dependency specification that you have used for *initialized setup*.

Exception handling

If an exception occurs while initializing or finalizing a driver, it will be bubbled up.

You can change this behaviour by providing an *on_exception* argument. It takes a callback with two arguments, the driver and the exception.

If you want to print the exception:

```
def print_and_continue(driver, ex):
    print('An exception occurred while initializing {}: {}'.format(driver, ex))

initialize_many(drivers, on_exception=print_and_continue)
```

or if you want to re-raise the exception, you can define a different callback:

```
def print_and_raise(driver, ex):
    print('An exception occurred while initializing {}: {}'.format(driver, ex))
    raise ex

initialize_many(drivers, on_exception=print_and_raise)
```

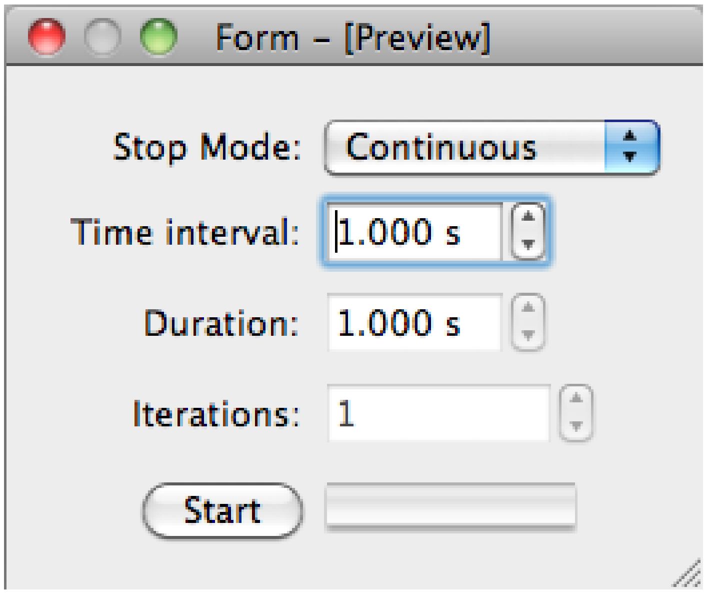
See also:

Initializing multiple drivers in a GUI

4.4.2 GUI (high-level)

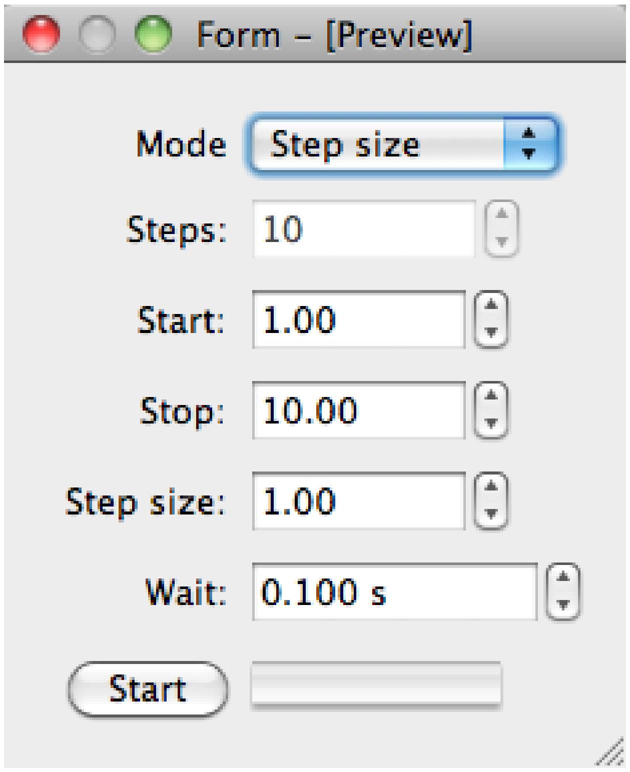
Knowing the Blocks

Loop


Form – [Preview]
Stop Mode: Continuous
Time interval: 1.000 s
Duration: 1.000 s
Iterations: 1

TBD

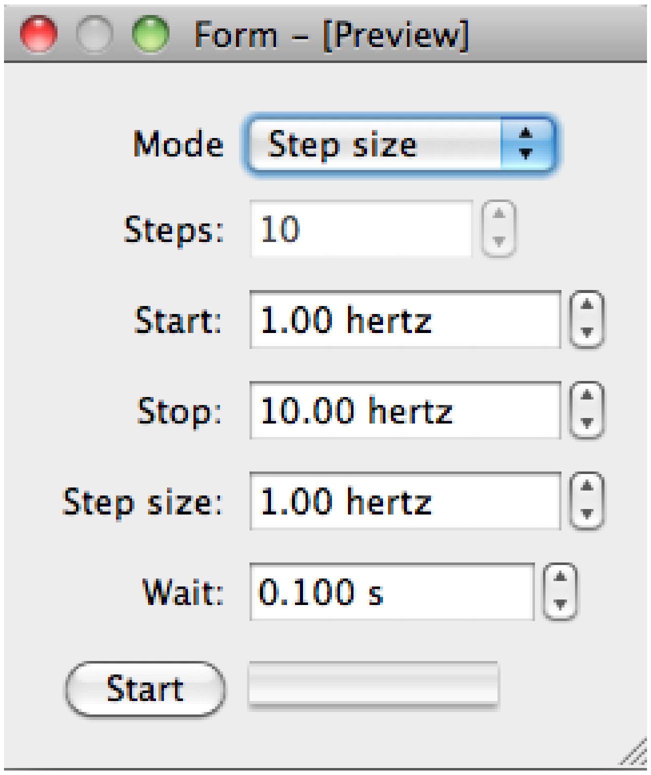
Scan



Mode	Step size
Steps:	10
Start:	1.00
Stop:	10.00
Step size:	1.00
Wait:	0.100 s

TBD

FeatScan


Form – [Preview]

Mode: Step size

Steps: 10

Start: 1.00 hertz

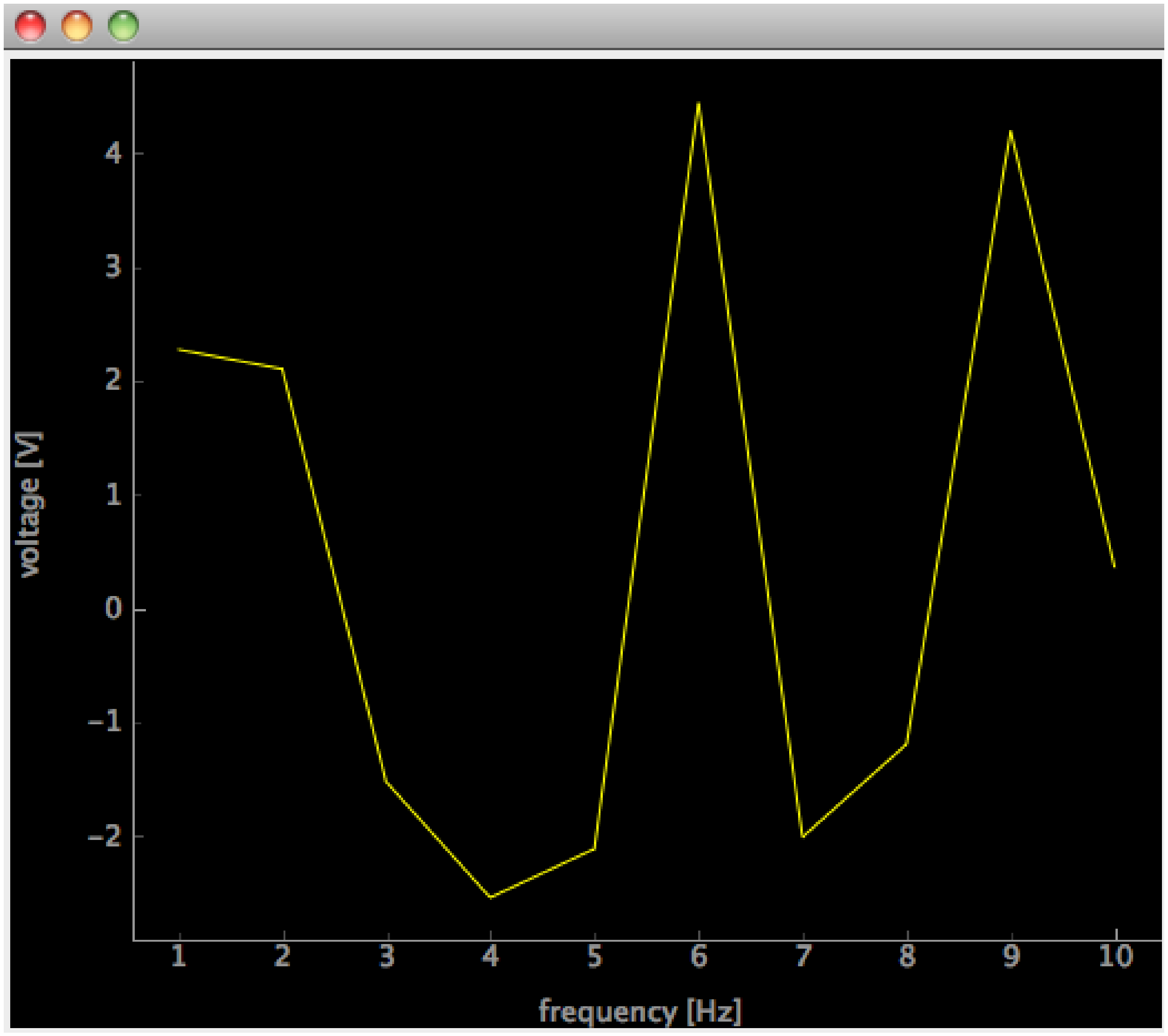
Stop: 10.00 hertz

Step size: 1.00 hertz

Wait: 0.100 s

TBD

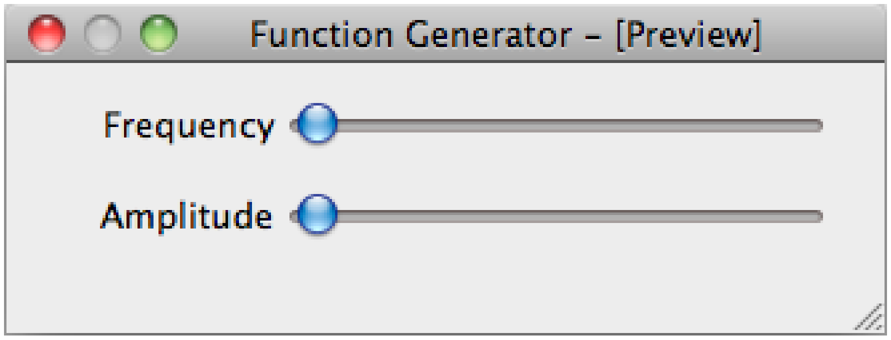
Chart



TBD

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*.

For *educational* purposes, we show you three ways to do this. You will certainly use only the last and shortest way but showing you how it is done allows you to understand what is going on.

The long way

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

```python
import sys

# Import Qt related from lantz so it worsk with PyQt4 or PySide ...
from lantz.utils.qt import QtGui

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

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

app = QtGui.QApplication(sys.argv)

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

# We get a reference to each of the widgets.
frequency_widget = main.findChild((QtGui.QWidget, ), 'frequency')
amplitude_widget = main.findChild((QtGui.QWidget, ), 'amplitude')

with LantzSignalGenerator('TCPIP::localhost::5678::SOCKET') 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):

```python
import sys

# Import Qt related from lantz so it worsk with PyQt4 or PySide ...
from lantz.utils.qt import QtGui

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

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

app = QtGui.QApplication(sys.argv)

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

with LantzSignalGenerator('TCPIP::localhost::5678::SOCKET') 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*.

The shortest way

As this is a commont pattern, we have a useful function for that:

```python
import sys

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

from lantz.ui.app import start_gui

with LantzSignalGenerator('TCPIP::localhost::5678::SOCKET') as inst:
    start_gui('connect_test.ui', inst, sys.argv)
```

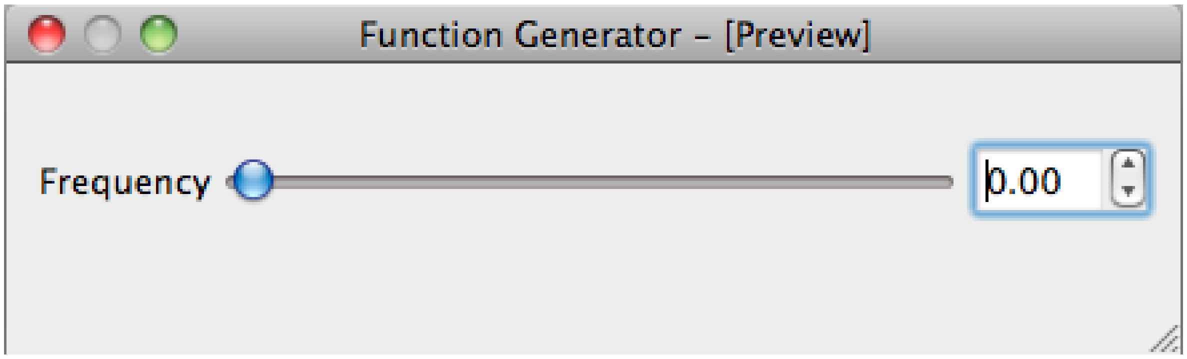
See also:

Connecting two (or more) widgets to the same feat

Connecting two (or more) drivers

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*.

For *educational* purposes, we show you three ways to do this. You will certainly use only the last and shortest way but showing you how it is done allows you to understand what is going on.

The long way

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

```
import sys

# Import Qt related from lantz so it worsk with PyQt4 or PySide ...
from lantz.utils.qt import QtGui

# 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.widgets import connect_feat

app = QtGui.QApplication(sys.argv)

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

# We get a reference to each of the widgets.
slider = main.findChild((QtGui.QWidget, ), 'frequency__slider')
spin = main.findChild((QtGui.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 Qt related from lantz so it worsk with PyQt4 or PySide ...
from lantz.utils.qt import QtGui

# 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.widgets import connect_feat

app = QtGui.QApplication(sys.argv)

# We load the UI from the QtDesigner file. You can also use pyuic4 to generate a class.
main = QtGui.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.

The shortest way

As this is a commont pattern, we have a useful function for that:

```
import sys

# Import Qt related from lantz so it worsk with PyQt4 or PySide ...
from lantz.ui.app import start_gui

with LantzSignalGeneratorTCP('localhost', 5678) as inst:
    start_gui('connect_test.ui', inst, sys.argv)
```

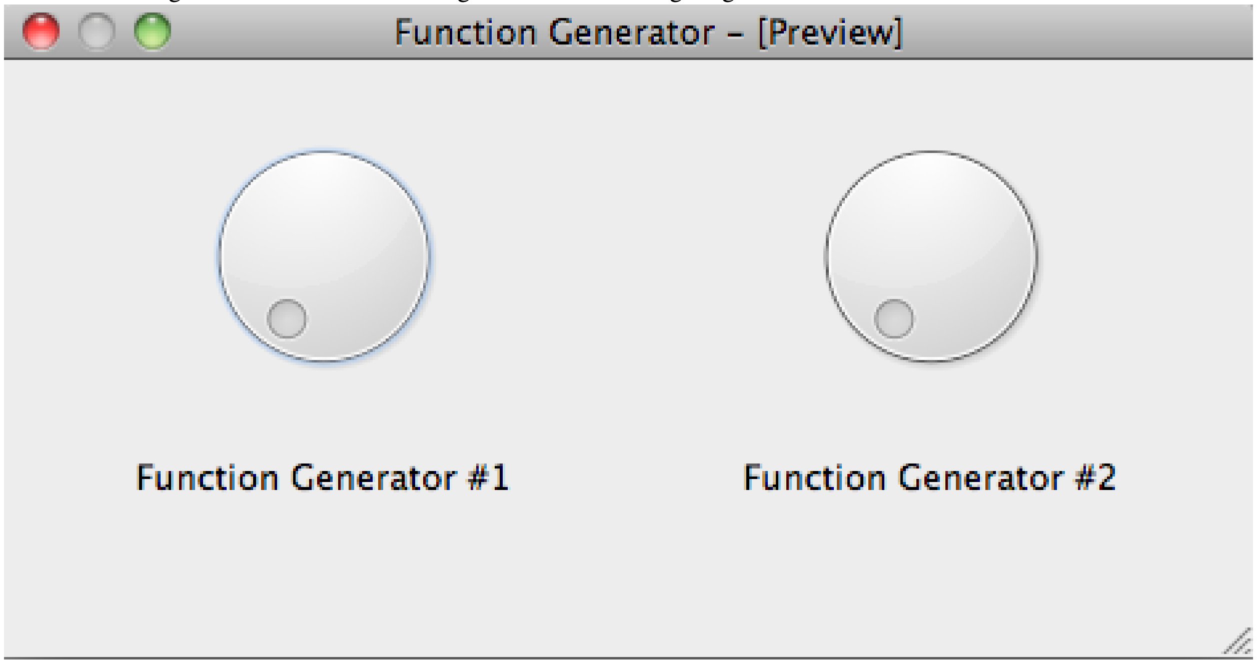
See also:

Connecting a custom UI to a driver

Connecting two (or more) drivers

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*.

For *educational* purposes, we show you four ways to do this. You will certainly use only the last and shortest way but showing you how it is done allows you to understand what is going on.

The long way

Get a reference to each widget and connect them manually:

```python
import sys

# Import Qt related from lantz so it worsk with PyQt4 or PySide ...
from lantz.utils.qt import QtGui

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

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

app = QtGui.QApplication(sys.argv)

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

# We get a reference to each of the widgets.
freq1 = main.findChild((QtGui.QWidget, ), 'fungen1__frequency')
freq2 = main.findChild((QtGui.QWidget, ), 'fungen2__frequency')
```

```
with LantzSignalGenerator('TCPIP::localhost::5678::SOCKET') as inst1, \
     LantzSignalGenerator('TCPIP::localhost::5678::SOCKET') 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 Qt related from lantz so it worsk with PyQt4 or PySide ...
from lantz.utils.qt import QtGui

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

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

app = QtGui.QApplication(sys.argv)

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

with LantzSignalGenerator('TCPIP::localhost::5678::SOCKET') as inst1, \
     LantzSignalGenerator('TCPIP::localhost::5678::SOCKET') 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 Qt related from lantz so it worsk with PyQt4 or PySide ...
from lantz.utils.qt import QtGui
```

```
# 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.widgets import connect_feat

app = QtGui.QApplication(sys.argv)

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

# Notice that now we specify the instrument name!
with LantzSignalGenerator('TCPIP::localhost::5678::SOCKET', name='fungen1') as inst1, \
     LantzSignalGenerator('TCPIP::localhost::5679::SOCKET', 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.

The shortest way

As this is a commont pattern, we have a useful function for that:

```
import sys


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

# Import Qt related from lantz so it worsk with PyQt4 or PySide ...
from lantz.ui.app import start_gui

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

    start_gui('connect_test.ui', [inst1, inst2], sys.argv)
```

See also:

Connecting a custom UI to a driver

Connecting two (or more) widgets to the same feat

Initializing multiple drivers in a GUI

A common part of an instrumentation GUI is to show a panel or a splash window to initialize instruments while providing some visual feedback about the status. Lantz provides a convenience method that can be hooked to different kind of widgets.

The function is called *initialize_and_report* and is defined in the widgets module:

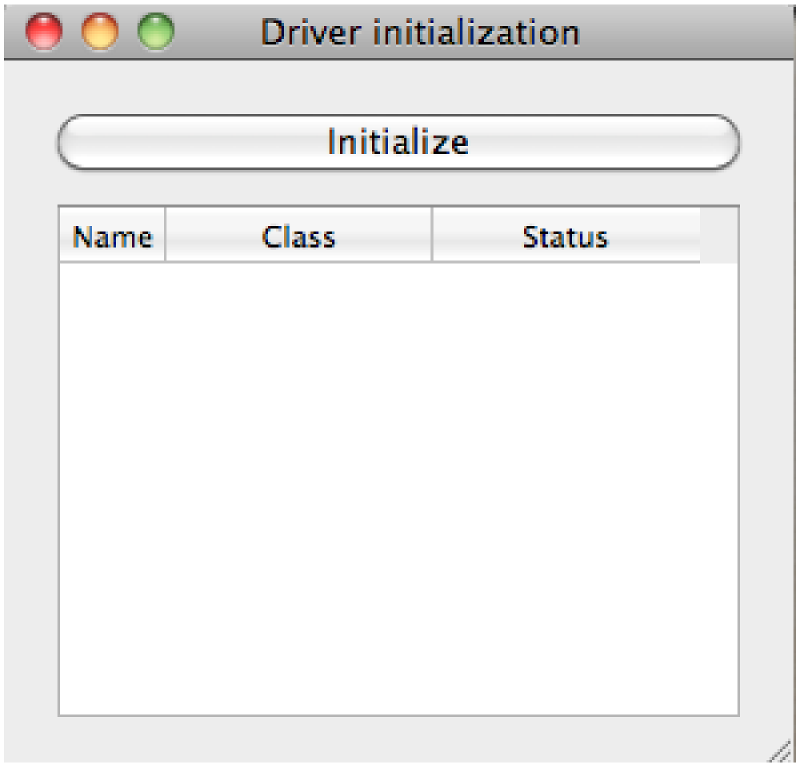
```
from lantz.ui.widgets import initialize_and_report
```

The signature of the function is very similar to *initialize_many*:

```
def initialize_and_report(widget, drivers, register_finalizer=True,
                          initializing_msg='Initializing ...', initialized_msg='Initialized',
                          concurrent=True, dependencies=None):
```

Supported widgets are: *QTableWidget* (useful for a modal window), *QLineEdit* (useful for a status bar), *QTextEdit* (useful for a log panel).

You can embed the widget in the window that you like. For example, in window like this:



The button initialize is connected to the following call:

```
initialize_and_report(status_widget, self.drivers)
```

that will start the process of initializing all drivers (in this case some are concurrently).

(see the code for this example https://github.com/hgrecco/lantz/blob/master/examples/gui_initializing.py)

See also:

Initializing multiple drivers

Building a Backend

TBD

Building a Frontend

TBD

4.4.3 Building drivers

The DEFAULTS dictionary

Different instruments require different communication settings such as baud rate, end of a message characters, etc. The :attribute::*DEFAULTS* dictionary provides a way to customize resource initialization at the :class::MessageBasedDriver level, avoiding tedious customization in all instances.

It is easier to see it with an example. Let's start with simple case:

```
class MyDriver(MessageBasedDriver):

    DEFAULTS = {
                  'COMMON':    {'write_termination': '\n'}
                }
```

The 'COMMON' key is used to tells MessageBasedDriver that 'write_termination' should be set to 'n' for all type of interface types (USB, GPIB, etc).

But in certain cases, different resource types might require different settings:

```
DEFAULTS = {
              'ASRL':    {'write_termination': '\n',
                          'read_termination': '\r',
                          'baud_rate': 9600},

              'USB':     {'write_termination': '\n',
                          'read_termination': \n'}
            }
```

This specifies a dictionary of settings for an ASRL (serial) resource and a different for USB. We might make this more concise:

```
DEFAULTS = {
              'ASRL':    {'read_termination': '\r',
                          'baud_rate': 9600},

              'USB':     {'read_termination': \n'},

              'COMMON':    {'write_termination': '\n'}
            }
```

When you require a USB resource, Lantz will combine the USB and COMMON settings.

The interface type is not the only thing that defines the resource. For example TCPIP device can be a INSTR or SOCKET. You can also specify this in a tuple:

```
DEFAULTS = {
              'INSTR':    {'read_termination': '\r'},

              'SOCKET':     {'read_termination': \n'},

              'COMMON':    {'write_termination': '\n'}
            }
```

This will specify that 'read_termination' will be set 'r' to for al INSTR. If you want to specify only for TCPIP, use a tuple like this:

```
DEFAULTS = {
              ('TCPIP, 'INSTR'):    {'read_termination': '\r'},
              'SOCKET':    {'read_termination': \n'},

              'COMMON':   {'write_termination': '\n'}
            }
```

Overriding on initialization

You can override the defaults when you instantiate the instrument by passing these values a command line arguments:

```
inst = MyDriver('TCPIP::localhost::5678::INSTR', read_termination='\t')
```

Colliding values

When multiple values are given for the same setting (for example 'read_termination' is in USB And COMMON) and a USB resource is requested, the following order is used to define the precedence:

- user provided keyword arguments.
- settings for (instrument_type, resource_type).
- settings for instrument_type: ASRL, USB, GPIB, TCPIP
- settings for resource_type: SOCKET, INSTR, RAW
- settings for COMMON

The rule is: more specific has precedence.

Valid settings

If you provide an invalid setting, you will get an Exception upon initalization. The valid settings are defined by Attributes per resource in PyVISA

4.4.4 GUI (low-level)

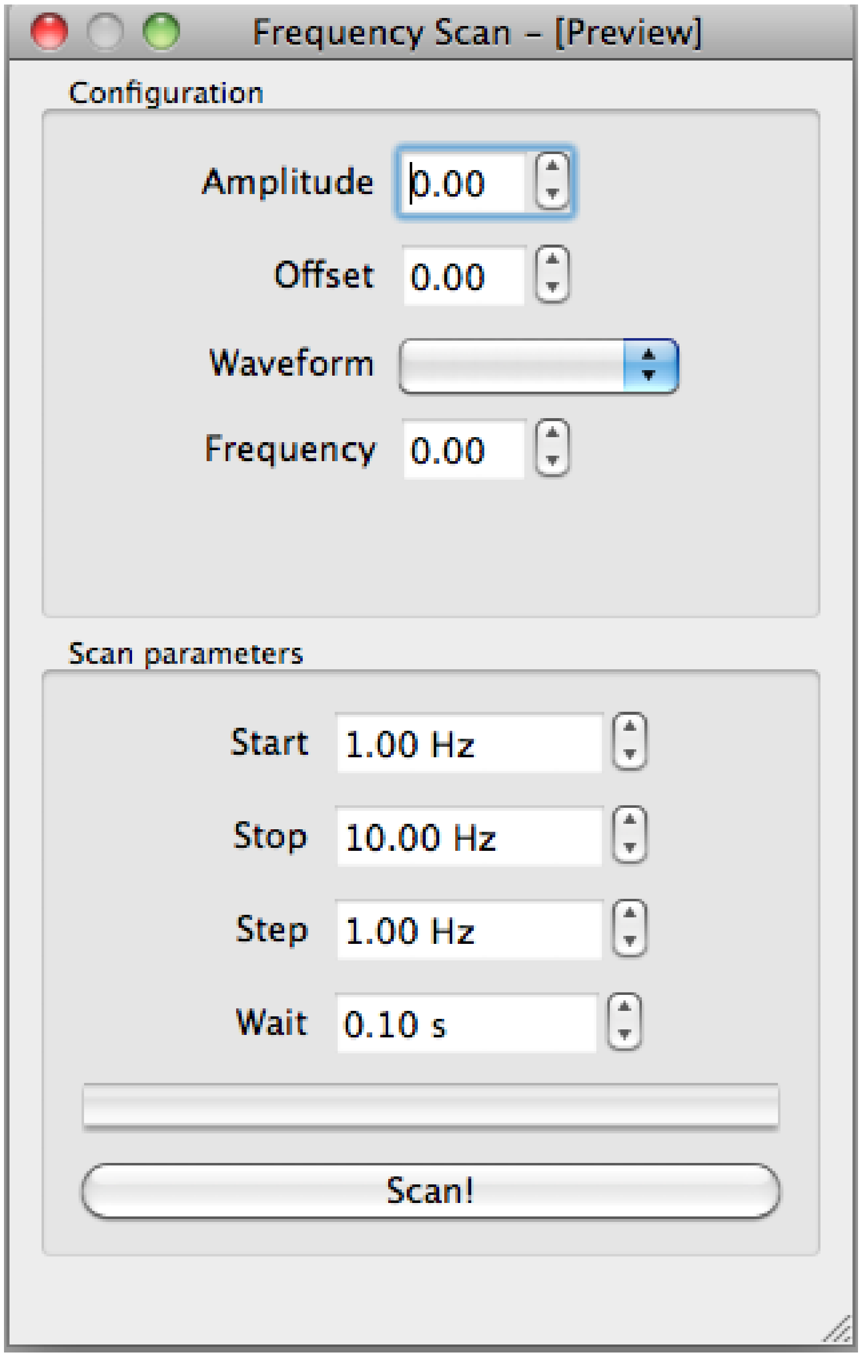
Building a GUI without Backend/Frontend

If you are a long time PyQt user, you might have asked yourself: Do I really need to use the :class:Backend and :class:Frontend classes? The answer is **No**. Lantz helps you, but only if you want it. You can do everything yourself and this guide shows you how.

Start the simulated instrument running the following command:

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

Using Qt Designer, create a window like this:



Amplitude 0.00
Offset 0.00
Frequency 0.00
Start 1.00 Hz
Stop 10.00 Hz
Step 1.00 Hz
Wait 0.10 s

and save it as *fungen.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.

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

```
import sys

from lantz import Q_

# Import Qt related from lantz so it worsk with PyQt4 or PySide ...
from lantz.utils.qt import QtGui
```

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

app = QtGui.QApplication(sys.argv)

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

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

with LantzSignalGenerator('TCPIP::localhost::5678::SOCKET') as inst:

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

from lantz import Q_

# Import Qt related from lantz so it worsk with PyQt4 or PySide ...
from lantz.utils.qt import QtGui

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

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

app = QtGui.QApplication(sys.argv)
```

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

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

with LantzSignalGenerator('TCPIP::localhost::5678::SOCKET') as inst:

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

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

    progress = main.findChild((QtGui.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 Qt related from lantz so it worsk with PyQt4 or PySide ...
from lantz.utils.qt import QtGui, QtCore

from lantz import Q_

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

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

app = QtGui.QApplication(sys.argv)

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

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

with LantzSignalGenerator('TCPIP::localhost::5678::SOCKET') a 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((QtGui.QWidget, ), 'start')
    stop = main.findChild((QtGui.QWidget, ), 'stop')
    step = main.findChild((QtGui.QWidget, ), 'step')
    wait = main.findChild((QtGui.QWidget, ), 'wait')
    scan = main.findChild((QtGui.QWidget, ), 'scan')
    progress = main.findChild((QtGui.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(QtCore.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 = QtCore.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.

:class:Backend and :class:Frontend provides an easier way to deal with all of these, allowing you to focus in your code not in how to connect everything.

4.4.5 Other guides

Upgrading to newer releases

Like any other package, Lantz is changing over time. Most of the changes are backwards compatible and you don't have to change anything in your code to enjoy the new release.

But every once in a while we to introduce some disruptive changes. We might realize that something error prone or the API was confusing. Or we might introduce a better way to do things and you might need to change your code to take advantage of it.

This section of the documentation enumerates all the backwards incompatible changes in Lantz, helping you to transition your code from release to release.

You can upgrade to the latest version of Lantz using pip:

```
pip install -U lantz
```

Upgrading to 0.3

We introduced the *lantz.messagebased.MessageBasedDriver*, a class to rule them all. Replaces `SerialDriver`, `TCPDriver`, `VisaDriver`, `USBDriver`, `USBTMCDriver`, `SerialVisaDriver`, `GPIBVisaDriver`, `USBVisaDriver`.

Migrating your driver to use *MessageBasedDriver* is easy:

1. Add the necessary imports:

```
    from lantz.messagebased import MessageBasedDriver
```

2. Change the base class of your driver:

```
    class MyDriver(MessageBasedDriver):

        # Your code
```

3. In the class methods, change *self.send* by *self.write*; and *self.recv* by *self.read*. This was done to homogenize the API with PyVISA
4. Change the class defaults to use the *The DEFAULTS dictionary*.

You are **NOT** forced to migrate you classes. The old base classes are still available under `lantz.drivers.legacy`. So you can just change the import.

While all instrument drivers have been migrated to the new `MessageBasedDriver` you can still used the drivers based on the legacy classes from `lantz.drivers.legacy`.

The only backwards incompatib

4.5 FAQs

4.5.1 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.

4.5.2 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.

4.5.3 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!

4.5.4 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.

4.5.5 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.

4.5.6 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.

4.5.7 How do I start?

The *Tutorials* is a good place.

4.5.8 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.

4.5.9 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.

4.6 Drivers

4.6.1 lantz.drivers.aa

company AA Opto Electronic.

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

website http://opto.braggcell.com/

---

copyright 2015 by Lantz Authors, see AUTHORS for more details.

license BSD, see LICENSE for more details.

class `lantz.drivers.aa.`**`MDSnC`**(*resource_name*, *name=None*, ***kwargs*)
Bases: *`lantz.messagebased.MessageBasedDriver`*

MDSnC synthesizer for AOTF.nC

Parameters

- **`resource_name`** (*`str`*) – The resource name
- **`kwargs`** – keyword arguments passed to the resource during initialization.

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

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

`CHANNELS_changed`

`enabled`

Keys [0, 1, 2, 3, 4, 5, 6, 7]

Enable single channels.

`enabled_changed`

`frequency`

Keys [0, 1, 2, 3, 4, 5, 6, 7]

RF frequency for a given channel.

`frequency_changed`

`main_enabled`
Enable the

Values {False: 0, True: 1}

`main_enabled_changed`

`power`

Keys [0, 1, 2, 3, 4, 5, 6, 7]

Power for a given channel (in digital units).

Limits (0, 1023, 1)

power_changed

powerdb

Keys [0, 1, 2, 3, 4, 5, 6, 7]

Power for a given channel (in db).

powerdb_changed

4.6.2 lantz.drivers.aeroflex

company Aeroflex

description Test and measurement equipment and microelectronic solutions.

website http://www.aeroflex.com/

---

copyright 2015 by Lantz Authors, see AUTHORS for more details.

license BSD, see LICENSE for more details.

class lantz.drivers.aeroflex.**A2023a**(*resource_name*, *name=None*, ***kwargs*)
Bases: *lantz.messagebased.MessageBasedDriver*

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

Parameters

- **resource_name** (*str*) – The resource name
- **kwargs** – keyword arguments passed to the resource during initialization.

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

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.

DEFAULTS = {'ASRL': {'read_termination': '', 'write_termination': '\n'}}

DEFAULTS_changed

`amplitude`
RF amplitude.

Units V

`amplitude_changed`

`clear_status` **= functools.partial(<bound method Action.call of <lantz.action.Action object at 0x7f2e5b877908>>, Non**

`clear_status_changed`

`event_status_enabled`
Standard event enable register.

`event_status_enabled_changed`

`event_status_reg`
Standard event enable register.

`event_status_reg_changed`

`expose` **= functools.partial(<bound method Action.call of <lantz.action.Action object at 0x7f2e5b87fe10>>, None)**

`expose_changed`

`fitted_options`
Fitted options.

`fitted_options_changed`

`frequency`
Carrier frequency.

Units Hz

`frequency_changed`

`frequency_standard`
Get internal or external frequency standard.

Values {‘EXT10IND’, ‘EXT10DIR’, ‘INT’, ‘EXTIND’, ‘INT10OUT’}

`frequency_standard_changed`

`idn`
Instrument identification.

`idn_changed`

`local_lockout_changed`

`offset`
Offset amplitude.

Units V

`offset_changed`

`output_enabled`
Enable or disable the RF output

Values {False: ‘DISABLED’, True: ‘ENABLED’}

`output_enabled_changed`

`phase`
Phase offset

Units deg

`phase_changed`

`remote_changed`

`reset` **= functools.partial(<bound method Action.call of <lantz.action.Action object at 0x7f2e5b871e48>>, None)**

`reset_changed`

`rflimit`
Set RF output level max.

`rflimit_changed`

`rflimit_enabled`

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

`rflimit_enabled_changed`

`self_test` **= functools.partial(<bound method Action.call of <lantz.action.Action object at 0x7f2e5b871eb8>>, None)**

`self_test_changed`

`service_request_enabled`
Service request enable register.

`service_request_enabled_changed`

`software_handshake_changed`

`status_byte`
Status byte, a number between 0-255.

`status_byte_changed`

`time`

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

`time_changed`

`trigger` **= functools.partial(<bound method Action.call of <lantz.action.Action object at 0x7f2e5b877128>>, None)**

`trigger_changed`

`wait` **= functools.partial(<bound method Action.call of <lantz.action.Action object at 0x7f2e5b877080>>, None)**

`wait_changed`

4.6.3 lantz.drivers.andor

company Andor

description Scientific cameras.

website http://www.andor.com/

---

copyright 2015 by Lantz Authors, see AUTHORS for more details.

license BSD, see LICENSE for more details.

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 ()
Finalize 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 ()
Initialize 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'

LIBRARY_NAME_changed

close = functools.partial(<bound method Action.call of <lantz.action.Action object at 0x7f2e5b809ef0>>, None)

close_changed

command_changed

finalize_changed

flush_changed

getbool_changed

getenumerated_changed

getfloat_changed

getint_changed

initialize_changed

is_implemented_changed

is_writable_changed

open = functools.partial(<bound method Action.call of <lantz.action.Action object at 0x7f2e5b809e48>>, None)

open_changed

queuebuffer_changed

setbool_changed

setenumerated_changed

setenumstring_changed

setfloat_changed

setint_changed

waitbuffer_changed

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
Pixel clock rate

Values {200: ‘200 MHz’, 280: ‘280 MHz’, 100: ‘100 MHz’}

clock_rate_changed

exposure_time
Get exposure time.

exposure_time_changed

fan_peed
Fan speed.

fan_peed_changed

initialize_changed

pixel_encoding
Pixel encoding.

Values {32: ‘Mono32’, 64: ‘Mono64’}

pixel_encoding_changed

roi
Set region of interest

roi_changed

sensor_size

sensor_size_changed

sensor_temp
Sensor temperature.

sensor_temp_changed

take_image = functools.partial(<bound method Action.call of <lantz.action.Action object at 0x7f2e5b74d198>>, None)

take_image_changed

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

QE(*wl*)
Returns the percentage QE for a particular head model at a user specified wavelength.

abort_acquisition_async(**args*, ***kwargs*)
(Async) This function aborts the current acquisition if one is active

acquired_data(*shape*)
This function will return the data from the last acquisition. The data are returned as long integers (32-bit signed integers). The “array” must be large enough to hold the complete data set.

acquired_data16(*shape*)
16-bit version of the GetAcquiredData function. The “array” must be large enough to hold the complete data set.

amp_available(*iamp*)
This function checks if the hardware and current settings permit the use of the specified amplifier.

amp_descr(*index*)
This function will return a string with an amplifier description. The amplifier is selected using the index. The SDK has a string associated with each of its amplifiers. The maximum number of characters needed to store the amplifier descriptions is 21. The user has to specify the number of characters they wish to have returned to them from this function.

amp_max_hspeed(*index*)
This function will return the maximum available horizontal shift speed for the amplifier selected by the index parameter.

bit_depth(*ch*)
This function will retrieve the size in bits of the dynamic range for any available AD channel.

camera_handle(*index*)
This function returns the handle for the camera specified by cameraIndex. When multiple Andor cameras are installed the handle of each camera must be retrieved in order to select a camera using the SetCurrent-Camera function. The number of cameras can be obtained using the GetAvailableCameras function.

Parameters **index** – index of any of the installed cameras. Valid values: 0 to NumberCameras-1 where NumberCameras is the value returned by the GetAvailableCameras function.

cancel_wait_async(**args*, ***kwargs*)
(Async) This function restarts a thread which is sleeping within the WaitForAcquisition function. The sleeping thread will return from WaitForAcquisition with a value not equal to DRV_SUCCESS.

`command` (*strcommand*)
Run command.

`count_convert_available` (*mode*)
This function checks if the hardware and current settings permit the use of the specified Count Convert mode.

`exposure_times` (*value*)

`finalize` ()
Finalize Library. Concluding function.

`flush` ()

`free_int_mem_async` (**args*, ***kwargs*)
(Async) The FreeInternalMemory function will deallocate any memory used internally to store the previously acquired data. Note that once this function has been called, data from last acquisition cannot be retrived.

`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.

`images` (*first*, *last*, *shape*, *validfirst*, *validlast*)
This function will update the data array with the specified series of images from the circular buffer. If the specified series is out of range (i.e. the images have been overwritten or have not yet been acquired) then an error will be returned.

Parameters

- **`first`** – index of first image in buffer to retrieve.
- **`flast`** – index of last image in buffer to retrieve.
- **`farr`** – pointer to data storage allocated by the user.
- **`size`** – total number of pixels.
- **`fvalidfirst`** – index of the first valid image.
- **`fvalidlast`** – index of the last valid image.

`images16` (*first*, *last*, *shape*, *validfirst*, *validlast*)
16-bit version of the GetImages function.

`initialize` ()
This function will initialize the Andor SDK System. As part of the initialization procedure on some cameras (i.e. Classic, iStar and earlier iXion) the DLL will need access to a DETECTOR.INI which contains information relating to the detector head, number pixels, readout speeds etc. If your system has multiple cameras then see the section Controlling multiple cameras.

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

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

most_recent_image(*shape*)
This function will update the data array with the most recently acquired image in any acquisition mode. The data are returned as long integers (32-bit signed integers). The "array" must be exactly the same size as the complete image.

most_recent_image16(*shape*)
16-bit version of the GetMostRecentImage function.

n_horiz_shift_speeds(*channel=0*, *typ=None*)
As your Andor SDK system is capable of operating at more than one horizontal shift speed this function will return the actual number of speeds available.

Parameters

- **channel** – the AD channel.
- **typ** – output amplification. 0 electron multiplication. 1 conventional.

oldest_image(*shape*)
This function will update the data array with the oldest image in the circular buffer. Once the oldest image has been retrieved it no longer is available. The data are returned as long integers (32-bit signed integers). The "array" must be exactly the same size as the full image.

oldest_image16(*shape*)
16-bit version of the GetOldestImage function.

preamp_available(*channel*, *amp*, *index*, *preamp*)
This function checks that the AD channel exists, and that the amplifier, speed and gain are available for the AD channel.

preamp_descr(*index*)
This function will return a string with a pre amp gain description. The pre amp gain is selected using the index. The SDK has a string associated with each of its pre amp gains. The maximum number of characters needed to store the pre amp gain descriptions is 30. The user has to specify the number of characters they wish to have returned to them from this function.

prepare_acquisition_async(**args*, ***kwargs*)
(Async) This function reads the current acquisition setup and allocates and configures any memory that will be used during the acquisition. The function call is not required as it will be called automatically by the StartAcquisition function if it has not already been called externally. However for long kinetic series acquisitions the time to allocate and configure any memory can be quite long which can result in a long delay between calling StartAcquisition and the acquisition actually commencing. For iDus, there is an additional delay caused by the camera being set-up with any new acquisition parameters. Calling PrepareAcquisition first will reduce this delay in the StartAcquisition call.

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

readout_flipped(*iamp*)
On cameras with multiple amplifiers the frame readout may be flipped. This function can be used to determine if this is the case.

save_raw_async(**args*, ***kwargs*)
(Async) This function saves the last acquisition as a raw data file. See self.savetypes for the file type keys.

send_software_trigger_async(**args*, ***kwargs*)
(Async) This function sends an event to the camera to take an acquisition when in Software Trigger mode. Not all cameras have this mode available to them. To check if your camera can operate in this mode check the GetCapabilities function for the Trigger Mode AC_TRIGGERMODE_CONTINUOUS. If this mode is physically possible and other settings are suitable (IsTriggerModeAvailable) and the camera is acquiring then this command will take an acquisition.

NOTES: The settings of the camera must be as follows: - ReadOut mode is full image - RunMode is Run Till Abort - TriggerMode is 10

sensitivity(*ad*, *amp*, *i*, *pa*)
This function returns the sensitivity for a particular speed.

set_accum_time_async(**args*, ***kwargs*)
(Async) This function will set the accumulation cycle time to the nearest valid value not less than the given value. The actual cycle time used is obtained by GetAcquisitionTimings. Please refer to SECTION 5 – ACQUISITION MODES for further information.

set_dma_parameters(*n_max_images*, *s_per_dma*)
In order to facilitate high image readout rates the controller card may wait for multiple images to be acquired before notifying the SDK that new data is available. Without this facility, there is a chance that hardware interrupts may be lost as the operating system does not have enough time to respond to each interrupt. The drawback to this is that you will not get the data for an image until all images for that interrupt have been acquired. There are 3 settings involved in determining how many images will be acquired for each notification (DMA Interrupt) of the controller card and they are as follows:

1. The size of the DMA buffer gives an upper limit on the number of images that can be stored within it and is usually set to the size of one full image when installing the software. This will usually mean that if you acquire full frames there will never be more than one image per DMA.

2. A second setting that is used is the minimum amount of time (SecondsPerDMA) that should expire between interrupts. This can be used to give an indication of the reponsiveness of the operating system to interrupts. Decreasing this value will allow more interrupts per second and should only be done for faster pcs. The default value is 0.03s (30ms), finding the optimal value for your pc can only be done through experimentation.

3. The third setting is an overide to the number of images calculated using the previous settings. If the number of images per dma is calculated to be greater than MaxImagesPerDMA then it will be reduced to MaxImagesPerDMA. This can be used to, for example, ensure that there is never more than 1 image per DMA by setting MaxImagesPerDMA to 1. Setting MaxImagesPerDMA to zero removes this limit. Care should be taken when modifying these parameters as missed interrupts may prevent the acquisition from completing.

set_exposure_time_async(**args*, ***kwargs*)
(Async) This function will set the exposure time to the nearest valid value not less than the given value, in seconds. The actual exposure time used is obtained by GetAcquisitionTimings. Please refer to SECTION 5 – ACQUISITION MODES for further information.

set_image_async(**args*, ***kwargs*)
(Async) This function will set the horizontal and vertical binning to be used when taking a full resolution image.

Parameters

- **hbin** – number of pixels to bin horizontally.
- **vbin** – number of pixels to bin vertically.
- **hstart** – Start column (inclusive).
- **hend** – End column (inclusive).
- **vstart** – Start row (inclusive).
- **vend** – End row (inclusive).

set_kinetic_cycle_time_async(**args*, ***kwargs*)
(Async) This function will set the kinetic cycle time to the nearest valid value not less than the given

value. The actual time used is obtained by GetAcquisitionTimings. . Please refer to SECTION 5 – ACQUISITION MODES for further information. float time: the kinetic cycle time in seconds.

set_n_accum_async (**args*, ***kwargs*)
(Async) This function will set the number of scans accumulated in memory. This will only take effect if the acquisition mode is either Accumulate or Kinetic Series.

set_n_kinetics_async (**args*, ***kwargs*)
(Async) This function will set the number of scans (possibly accumulated scans) to be taken during a single acquisition sequence. This will only take effect if the acquisition mode is Kinetic Series.

set_photon_counting_divs_async (**args*, ***kwargs*)
(Async) This function sets the thresholds for the photon counting option.

set_photon_counting_thres_async (**args*, ***kwargs*)
(Async) This function sets the minimum and maximum threshold in counts (1-65535) for the photon counting option.

set_vert_clock_async (**args*, ***kwargs*)
(Async) If you choose a high readout speed (a low readout time), then you should also consider increasing the amplitude of the Vertical Clock Voltage. There are five levels of amplitude available for you to choose from: - Normal, +1, +2, +3, +4 Exercise caution when increasing the amplitude of the vertical clock voltage, since higher clocking voltages may result in increased clock-induced charge (noise) in your signal. In general, only the very highest vertical clocking speeds are likely to benefit from an increased vertical clock voltage amplitude.

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.

shutter_async (**args*, ***kwargs*)
(Async) This function expands the control offered by SetShutter to allow an external shutter and internal shutter to be controlled independently (only available on some cameras – please consult your Camera User Guide). The typ parameter allows the user to control the TTL signal output to an external shutter. The opening and closing times specify the length of time required to open and close the shutter (this information is required for calculating acquisition timings – see SHUTTER TRANSFER TIME). The mode and extmode parameters control the behaviour of the internal and external shutters. To have an external shutter open and close automatically in an experiment, set the mode parameter to "Open" and set the extmode parameter to "Auto". To have an internal shutter open and close automatically in an experiment, set the extmode parameter to "Open" and set the mode parameter to "Auto". To not use any shutter in the experiment, set both shutter modes to permanently open.

Parameters

- **typ** – 0 (or 1) Output TTL low (or high) signal to open shutter.
- **mode** – Internal shutter: 0 Fully Auto, 1 Permanently Open, 2 Permanently Closed, 4 Open for FVB series, 5 Open for any series.
- **ext_closing** – Time shutter takes to close (milliseconds)

- **ext_opening** – Time shutter takes to open (milliseconds)
- **ext_mode** – External shutter: 0 Fully Auto, 1 Permanently Open, 2 Permanently Closed, 4 Open for FVB series, 5 Open for any series.

start_acquisition_async(**args*, ***kwargs*)
(Async) This function starts an acquisition. The status of the acquisition can be monitored via GetStatus().

trigger_level_async(**args*, ***kwargs*)
(Async) This function sets the trigger voltage which the system will use.

trigger_mode_available(*modestr*)
This function checks if the hardware and current settings permit the use of the specified trigger mode.

true_exposure_times(*n*)
This function will return the actual exposure times that the camera will use. There may be differences between requested exposures and the actual exposures. ntimes: Numbers of times requested.

true_horiz_shift_speed(*index=0*, *typ=None*, *ad=0*)
As your Andor system is capable of operating at more than one horizontal shift speed this function will return the actual speeds available. The value returned is in MHz.

GetHSSpeed(int channel, int typ, int index, float* speed)

Parameters

- **typ** – output amplification. 0 electron multiplication/Conventional(clara) 1 conventional/Extended NIR Mode(clara).
- **index** – speed required 0 to NumberSpeeds-1 where NumberSpeeds is value returned in first parameter after a call to GetNumberHSSpeeds().
- **ad** – the AD channel.

true_preamp(*index*)
For those systems that provide a number of pre amp gains to apply to the data as it is read out; this function retrieves the amount of gain that is stored for a particular index. The number of gains available can be obtained by calling the GetNumberPreAmpGains function and a specific Gain can be selected using the function SetPreAmpGain.

true_vert_amp(*index*)
This Function is used to get the value of the Vertical Clock Amplitude found at the index passed in.

Parameters **index** – Index of VS amplitude required Valid values 0 to GetNumberVSAmplitudes() - 1

true_vert_shift_speed(*index=0*)
As your Andor SDK system may be capable of operating at more than one vertical shift speed this function will return the actual speeds available. The value returned is in microseconds.

vert_amp_index(*string*)
This Function is used to get the index of the Vertical Clock Amplitude that corresponds to the string passed in.

Parameters **string** – "Normal" , "+1" , "+2" , "+3" , "+4"

vert_amp_string(*index*)
This Function is used to get the Vertical Clock Amplitude string that corresponds to the index passed in.

Parameters **index** – Index of VS amplitude required Valid values 0 to GetNumberVSAmplitudes() - 1

wait_for_acquisition_async(**args*, ***kwargs*)
(Async) WaitForAcquisition can be called after an acquisition is started using StartAcquisition to put the

calling thread to sleep until an Acquisition Event occurs. This can be used as a simple alternative to the functionality provided by the SetDriverEvent function, as all Event creation and handling is performed internally by the SDK library. Like the SetDriverEvent functionality it will use less processor resources than continuously polling with the GetStatus function. If you wish to restart the calling thread without waiting for an Acquisition event, call the function CancelWait. An Acquisition Event occurs each time a new image is acquired during an Accumulation, Kinetic Series or Run-Till-Abort acquisition or at the end of a Single Scan Acquisition. If a second event occurs before the first one has been acknowledged, the first one will be ignored. Care should be taken in this case, as you may have to use CancelWait to exit the function.

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

`EM_advanced_enabled`

This function turns on and off access to higher EM gain levels within the SDK. Typically, optimal signal to noise ratio and dynamic range is achieved between x1 to x300 EM Gain. Higher gains of > x300 are recommended for single photon counting only. Before using higher levels, you should ensure that light levels do not exceed the regime of tens of photons per pixel, otherwise accelerated ageing of the sensor can occur. This is set to False upon initialization of the camera.

Values {False: 0, True: 1}

`EM_advanced_enabled_changed`

`EM_gain`
Allows the user to change the gain value. The valid range for the gain depends on what gain mode the camera is operating in. See SetEMGainMode to set the mode and GetEMGainRange to get the valid range to work with. To access higher gain values (>x300) see SetEMAdvanced.

`EM_gain_changed`

`EM_gain_mode`

Set the EM Gain mode to one of the following possible settings. Mode 0: The EM Gain is controlled by DAC settings in the range 0-255. Default mode. 1: The EM Gain is controlled by DAC settings in the range 0-4095. 2: Linear mode. 3: Real EM gain

Values {'DAC255': 0, 'DAC4095': 1, 'Linear': 2, 'RealGain': 3}

`EM_gain_mode_changed`

`EM_gain_range`
Returns the minimum and maximum values of the current selected EM Gain mode and temperature of the sensor.

`EM_gain_range_changed`

`FK_exposure_time`

This function will return the current "valid" exposure time for a fast kinetics acquisition. This function should be used after all the acquisitions settings have been set, i.e. SetFastKinetics and SetFKVShiftSpeed. The value returned is the actual time used in subsequent acquisitions.

Units s

`FK_exposure_time_changed`

`LIBRARY_NAME` = **'atmcd64d.dll'**

LIBRARY_NAME_changed

QE_changed

abort_acquisition = functools.partial(<bound method Action.call of <lantz.action.Action object at 0x7f2e5b6cbf60>

abort_acquisition_changed

acquired_data16_changed

acquired_data_changed

acquisition_mode

This function will set the acquisition mode to be used on the next StartAcquisition. NOTE: In Mode 5 the system uses a "Run Till Abort" acquisition mode. In Mode 5 only, the camera continually acquires data until the AbortAcquisition function is called. By using the SetDriverEvent function you will be notified as each acquisition is completed.

Values {'Kinetics': 3, 'Fast Kinetics': 4, 'Run till abort': 5, 'Accumulate': 2, 'Single Scan': 1}

acquisition_mode_changed

acquisition_progress

This function will return information on the progress of the current acquisition. It can be called at any time but is best used in conjunction with SetDriverEvent. The values returned show the number of completed scans in the current acquisition. If 0 is returned for both accum and series then either: - No acquisition is currently running - The acquisition has just completed - The very first scan of an acquisition has just started and not yet completed. GetStatus can be used to confirm if the first scan has just started, returning DRV_ACQUIRING, otherwise it will return DRV_IDLE. For example, if accum=2 and series=3 then the acquisition has completed 3 in the series and 2 accumulations in the 4 scan of the series

acquisition_progress_changed

acquisition_timings

This function will return the current "valid" acquisition timing information. This function should be used after all the acquisitions settings have been set, e.g. SetExposureTime, SetKineticCycleTime and SetReadMode etc. The values returned are the actual times used in subsequent acquisitions. This function is required as it is possible to set the exposure time to 20ms, accumulate cycle time to 30ms and then set the readout mode to full image. As it can take 250ms to read out an image it is not possible to have a cycle time of 30ms. All data is measured in seconds.

acquisition_timings_changed

adv_trigger_mode

This function will set the state for the iCam functionality that some cameras are capable of. There may be some cases where we wish to prevent the software using the new functionality and just do it the way it was previously done.

Values {False: 0, True: 1}

adv_trigger_mode_changed

amp_available_changed

amp_descr_changed

amp_max_hspeed_changed

`available_images_index`
This function will return information on the number of available images in the circular buffer. This information can be used with GetImages to retrieve a series of images. If any images are overwritten in the circular buffer they no longer can be retrieved and the information returned will treat overwritten images as not available.

`available_images_index_changed`

`averaging_factor`
Averaging factor to be used with the recursive filter. For information on the various data averaging filters available see DATA AVERAGING FILTERS in the Special Guides section of the manual.

`averaging_factor_changed`

`averaging_frame_count`
Number of frames to be used when using the frame averaging filter.

`averaging_frame_count_changed`

`averaging_mode`

Current averaging mode. Valid options are: 0 – No Averaging Filter 5 – Recursive Averaging Filter 6 – Frame Averaging Filter

Values {‘FAF’: 6, ‘NAF’: 0, ‘RAF’: 5}

`averaging_mode_changed`

`baseline_clamp`

This function returns the status of the baseline clamp functionality. With this feature enabled the baseline level of each scan in a kinetic series will be more consistent across the sequence.

Values {False: 0, True: 1}

`baseline_clamp_changed`

`baseline_offset`

This function allows the user to move the baseline level by the amount selected. For example “+100” will add approximately 100 counts to the default baseline value. The value entered should be a multiple of 100 between -1000 and +1000 inclusively.

Limits (-1000, 1100, 100)

`baseline_offset_changed`

`bit_depth_changed`

`buffer_size`
This function will return the maximum number of images the circular buffer can store based on the current acquisition settings.

`buffer_size_changed`

`camera_handle_changed`

`cancel_wait` = functools.partial(<bound method Action.call of <lantz.action.Action object at 0x7f2e5b6d10f0>>, None)

`cancel_wait_changed`

capabilities
This function will fill in an AndorCapabilities structure with the capabilities associated with the connected camera. Individual capabilities are determined by examining certain bits and combinations of bits in the member variables of the AndorCapabilites structure.

capabilities_changed

command_changed

controller_card
This function will retrieve the type of PCI controller card included in your system. This function is not applicable for USB systems. The maximum number of characters that can be returned from this function is 10.

controller_card_changed

cooled_on_shutdown

This function determines whether the cooler is switched off when the camera is shut down.

Values {False: 0, True: 1}

cooled_on_shutdown_changed

cooler_on

Values {False: 0, True: 1}

cooler_on_changed

count_convert_available_changed

count_convert_wavelength_range
This function returns the valid wavelength range available in Count Convert mode.

count_convert_wavelength_range_changed

cr_filter_enabled

This function will set the state of the cosmic ray filter mode for future acquisitions. If the filter mode is on, consecutive scans in an accumulation will be compared and any cosmic ray-like features that are only present in one scan will be replaced with a scaled version of the corresponding pixel value in the correct scan.

Values {False: 0, True: 2}

cr_filter_enabled_changed

current_camera
When multiple Andor cameras are installed this function allows the user to select which camera is currently active. Once a camera has been selected the other functions can be called as normal but they will only apply to the selected camera. If only 1 camera is installed calling this function is not required since that camera will be selected by default.

current_camera_changed

detector_shape

detector_shape_changed

exposing

This function will return if the system is exposing or not. The status of the firepulse will be returned. NOTE This is only supported by the CCI23 card.

Values {False: 0, True: 1}

`exposing_changed`

`exposure_times_changed`

`fan_mode`

Allows the user to control the mode of the camera fan. If the system is cooled, the fan should only be turned off for short periods of time. During this time the body of the camera will warm up which could compromise cooling capabilities. If the camera body reaches too high a temperature, depends on camera, the buzzer will sound. If this happens, turn off the external power supply and allow the system to stabilize before continuing.

Values {'onlow': 1, 'onfull': 0, 'off': 2}

`fan_mode_changed`

`fastest_recommended_vsspeed`
As your Andor SDK system may be capable of operating at more than one vertical shift speed this function will return the fastest recommended speed available. The very high readout speeds, may require an increase in the amplitude of the Vertical Clock Voltage using SetVSAmplitude. This function returns the fastest speed which does not require the Vertical Clock Voltage to be adjusted. The values returned are the vertical shift speed index and the actual speed in microseconds per pixel shift.

`fastest_recommended_vsspeed_changed`

`filter_threshold`
Sets the threshold value for the Noise Filter. For information on the various spurious noise filters available see SPURIOUS NOISE FILTERS in the Special Guides section of the manual. Valid values are: 0 – 65535 for Level Above filte 0 – 10 for all other filters.

`filter_threshold_changed`

`finalize_changed`

`flush_changed`

`frame_transfer_mode`

This function will set whether an acquisition will readout in Frame Transfer Mode. If the acquisition mode is Single Scan or Fast Kinetics this call will have no affect.

Values {False: 0, True: 1}

`frame_transfer_mode_changed`

`free_int_mem` = functools.partial(<bound method Action.call of <lantz.action.Action object at 0x7f2e5b6da4e0>>, Non

`free_int_mem_changed`

`getbool_changed`

`getenumerated_changed`

`getfloat_changed`

`getint_changed`

`hardware_version`

`hardware_version_changed`

`has_mechanical_shutter`

`has_mechanical_shutter_changed`

`horiz_shift_speed`
This function will set the speed at which the pixels are shifted into the output node during the readout phase of an acquisition. Typically your camera will be capable of operating at several horizontal shift speeds. To get the actual speed that an index corresponds to use the GetHSSpeed function.

Parameters

- **`typ`** – output amplification. 0 electron multiplication/Conventional(clara). 1 conventional/Extended NIR mode(clara).
- **`index`** – the horizontal speed to be used 0 to GetNumberHSSpeeds() - 1

`horiz_shift_speed_changed`

`idn`
Identification of the device

`idn_changed`

`images16_changed`

`images_changed`

`in_aux_port`

Keys [1, 2, 3, 4]

This function returns the state of the TTL Auxiliary Input Port on the Andor plug-in card.

Values {False: 0, True: True}

`in_aux_port_changed`

`initialize_changed`

`is_implemented_changed`

`is_writable_changed`

`keep_clean_time`

This function will return the time to perform a keep clean cycle. This function should be used after all the acquisitions settings have been set, e.g. SetExposureTime, SetKineticCycleTime and SetReadMode etc. The value returned is the actual times used in subsequent acquisitions.

Units s

`keep_clean_time_changed`

`max_exposure`

This function will return the maximum Exposure Time in seconds that is settable by the SetExposureTime function.

Units s

`max_exposure_changed`

`max_images_per_dma`
This function will return the maximum number of images that can be transferred during a single DMA transaction.

`max_images_per_dma_changed`

`max_temperature`

This function returns the valid range of temperatures in centigrads to which the detector can be cooled.

Units degC

`max_temperature_changed`

`min_image_length`
This function will return the minimum number of pixels that can be read out from the chip at each exposure. This minimum value arises due the way in which the chip is read out and will limit the possible sub image dimensions and binning sizes that can be applied.

`min_image_length_changed`

`min_temperature`

This function returns the valid range of temperatures in centigrads to which the detector can be cooled.

Units degC

`min_temperature_changed`

`most_recent_image16_changed`

`most_recent_image_changed`

`n_ad_channels`

`n_ad_channels_changed`

`n_amps`

`n_amps_changed`

`n_exposures_in_ring`
Gets the number of exposures in the ring at this moment.

`n_exposures_in_ring_changed`

`n_horiz_shift_speeds_changed`

`n_images_acquired`
This function will return the total number of images acquired since the current acquisition started. If the camera is idle the value returned is the number of images acquired during the last acquisition.

`n_images_acquired_changed`

`n_max_nexposure`
This function will return the maximum number of exposures that can be configured in the SetRingExposureTimes SDK function.

`n_max_nexposure_changed`

`n_photon_counting_div`
Available in some systems is photon counting mode. This function gets the number of photon counting divisions available. The functions SetPhotonCounting and SetPhotonCountingThreshold can be used to specify which of these divisions is to be used.

`n_photon_counting_div_changed`

`n_preamps`
Available in some systems are a number of pre amp gains that can be applied to the data as it is read out. This function gets the number of these pre amp gains available. The functions GetPreAmpGain and SetPreAmpGain can be used to specify which of these gains is to be used.

`n_preamps_changed`

`n_vert_clock_amps`
This function will normally return the number of vertical clock voltage amplitudes that the camera has.

`n_vert_clock_amps_changed`

`n_vert_shift_speeds`
As your Andor system may be capable of operating at more than one vertical shift speed this function will return the actual number of speeds available.

`n_vert_shift_speeds_changed`

`ncameras`
This function returns the total number of Andor cameras currently installed. It is possible to call this function before any of the cameras are initialized.

`ncameras_changed`

`new_images_index`
This function will return information on the number of new images (i.e. images which have not yet been retrieved) in the circular buffer. This information can be used with GetImages to retrieve a series of the latest images. If any images are overwritten in the circular buffer they can no longer be retrieved and the information returned will treat overwritten images as having been retrieved.

`new_images_index_changed`

`noise_filter_mode`

Set the Noise Filter to use; For information on the various spurious noise filters available see SPURIOUS NOISE FILTERS in the Special Guides section of the manual. Valid options are: 0 – No Averaging Filter 1 – Median Filter 2 – Level Above Filter 3 – Interquartile Range Filter 4 – Noise Threshold Filter

Values {‘NF’: 0, ‘MF’: 1, ‘NTF’: 4, ‘LAF’: 2, ‘IRF’: 3}

`noise_filter_mode_changed`

`oldest_image16_changed`

`oldest_image_changed`

`out_aux_port`

Keys [1, 2, 3, 4]

This function sets the TTL Auxiliary Output port (P) on the Andor plug-in card to either ON/HIGH or OFF/LOW.

Values {False: 0, True: 1}

`out_aux_port_changed`

`photon_counting_mode`
This function activates the photon counting option.

Values {False: 0, True: 1}

photon_counting_mode_changed

preamp
This function will set the pre amp gain to be used for subsequent acquisitions. The actual gain factor that will be applied can be found through a call to the GetPreAmpGain function. The number of Pre Amp Gains available is found by calling the GetNumberPreAmpGains function.

preamp_available_changed

preamp_changed

preamp_descr_changed

prepare_acquisition = functools.partial(<bound method Action.call of <lantz.action.Action object at 0x7f2e5b6cbda

prepare_acquisition_changed

px_size
This function returns the dimension of the pixels in the detector in microns.

px_size_changed

queuebuffer_changed

readout_flipped_changed

readout_mode

This function will set the readout mode to be used on the subsequent acquisitions.

Values {'Single-Track': 3, 'FVB': 0, 'Image': 4, 'Multi-Track': 1, 'Random-Track': 2}

readout_mode_changed

readout_packing

This function will configure whether data is packed into the readout register to improve frame rates for sub-images. Note: It is important to ensure that no light falls outside of the sub-image area otherwise the acquired data will be corrupted. Only currently available on iXon+ and iXon3.

Values {False: 0, True: 1}

readout_packing_changed

readout_time

This function will return the time to readout data from a sensor. This function should be used after all the acquisitions settings have been set, e.g. SetExposureTime, SetKineticCycleTime and SetReadMode etc. The value returned is the actual times used in subsequent acquisitions.

Units s

readout_time_changed

save_raw = functools.partial(<bound method Action.call of <lantz.action.Action object at 0x7f2e5b6dab70>>, None)

save_raw_changed

send_software_trigger = functools.partial(<bound method Action.call of <lantz.action.Action object at 0x7f2e5b6a

send_software_trigger_changed

sensitivity_changed

set_accum_time = functools.partial(<bound method Action.call of <lantz.action.Action object at 0x7f2e5b6dae80>>, N

set_accum_time_changed

set_dma_parameters_changed

set_exposure_time = functools.partial(<bound method Action.call of <lantz.action.Action object at 0x7f2e5b6dae10>

set_exposure_time_changed

set_image = functools.partial(<bound method Action.call of <lantz.action.Action object at 0x7f2e5b6d1d30>>, None)

set_image_changed

set_kinetic_cycle_time = functools.partial(<bound method Action.call of <lantz.action.Action object at 0x7f2e5b6

set_kinetic_cycle_time_changed

set_n_accum = functools.partial(<bound method Action.call of <lantz.action.Action object at 0x7f2e5b662198>>, None)

set_n_accum_changed

set_n_kinetics = functools.partial(<bound method Action.call of <lantz.action.Action object at 0x7f2e5b6620f0>>, N

set_n_kinetics_changed

set_photon_counting_divs = functools.partial(<bound method Action.call of <lantz.action.Action object at 0x7f2e5

set_photon_counting_divs_changed

set_photon_counting_thres = functools.partial(<bound method Action.call of <lantz.action.Action object at 0x7f2

set_photon_counting_thres_changed

set_vert_clock = functools.partial(<bound method Action.call of <lantz.action.Action object at 0x7f2e5b671438>>, N

set_vert_clock_changed

setbool_changed

setenumerated_changed

setenumstring_changed

setfloat_changed

setint_changed

shutter = functools.partial(<bound method Action.call of <lantz.action.Action object at 0x7f2e5b6b3da0>>, None)

shutter_changed

shutter_min_times

This function will return the minimum opening and closing times in milliseconds for the shutter on the current camera.

shutter_min_times_changed

software_version

software_version_changed

start_acquisition = functools.partial(<bound method Action.call of <lantz.action.Action object at 0x7f2e5b6cbe10>

start_acquisition_changed

status

This function will return the current status of the Andor SDK system. This function should be called before an acquisition is started to ensure that it is IDLE and during an acquisition to monitor the process.

status_changed

`temperature`

This function returns the temperature of the detector to the nearest degree. It also gives the status of cooling process.

Units degC

`temperature_changed`

`temperature_setpoint`

Units degC

`temperature_setpoint_changed`

`temperature_status`

This function returns the temperature of the detector to the nearest degree. It also gives the status of cooling process.

`temperature_status_changed`

`trigger_level` **= functools.partial(<bound method Action.call of <lantz.action.Action object at 0x7f2e5b6a7e10>>, No**

`trigger_level_changed`

`trigger_mode`

This function will set the trigger mode that the camera will operate in.

Values {'Software Trigger': 10, 'External': 1, 'Internal': 0, 'External FVB EM': 9, 'External Start': 6, 'External Exposure': 7, 'External Charge Shifting': 12}

`trigger_mode_available_changed`

`trigger_mode_changed`

`true_exposure_times_changed`

`true_horiz_shift_speed_changed`

`true_preamp_changed`

`true_vert_amp_changed`

`true_vert_shift_speed_changed`

`vert_amp_index_changed`

`vert_amp_string_changed`

`vert_shift_speed`

This function will set the vertical speed to be used for subsequent acquisitions.

`vert_shift_speed_changed`

`wait_for_acquisition` **= functools.partial(<bound method Action.call of <lantz.action.Action object at 0x7f2e5b6d1**

`wait_for_acquisition_changed`

`waitbuffer_changed`

4.6.4 lantz.drivers.cobolt

company Cobolt.

description DPSS lasers, diode laser modules, fiber pigtailed lasers.

website http://www.cobolt.se/

---

copyright 2015 by Lantz Authors, see AUTHORS for more details.

license BSD, see LICENSE for more details.

class `lantz.drivers.cobolt.`**`Cobolt0601`**(*resource_name*, *name=None*, ***kwargs*)
Bases: *`lantz.messagebased.MessageBasedDriver`*

Driver for any Cobolt 06-01 Series laser.

Parameters

- **`resource_name`** (*`str`*) – The resource name
- **`kwargs`** – keyword arguments passed to the resource during initialization.

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

`clear_fault_async`(**args*, ***kwargs*)
(Async) Clear fault

`enter_mod_mode_async`(**args*, ***kwargs*)
(Async) Enter modulation mode

`initialize`()

`restart_async`(**args*, ***kwargs*)
(Async) Forces the laser on without checking if autostart is enabled.

`DEFAULTS` = {'ASRL': {'read_termination': '\r', 'parity': <Parity.none: 0>, 'baud_rate': 115200, 'bytesize': 8, 'stop_bits

`DEFAULTS_changed`

`analog_mod`
analog modulation enable state

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

`analog_mod_changed`

`analogli_mod`
analog modulation enable state

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

`analogli_mod_changed`

`autostart`
Autostart handling

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

`autostart_changed`

`clear_fault` = functools.partial(<bound method Action.call of <lantz.action.Action object at 0x7f2e5b415d30>>, None)

`clear_fault_changed`

`ctl_mode`
To handle laser control modes

Values {'ACC', 'APC'}

`ctl_mode_changed`

`current_sp`
Get drive current

Units mA

`current_sp_changed`

`digital_mod`
digital modulation enable state

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

`digital_mod_changed`

`enabled`
Method for turning on the laser. Requires autostart disabled.

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

`enabled_changed`

`enter_mod_mode` = functools.partial(<bound method Action.call of <lantz.action.Action object at 0x7f2e5b415da0>>, N

`enter_mod_mode_changed`

`idn`
Get serial number

`idn_changed`

`initialize_changed`

`interlock`
Get interlock state

Values {'Interlock open': '1', 'OK': '0'}

`interlock_changed`

`ksw_enabled`
Handling Key Switch enable state

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

`ksw_enabled_changed`

`mod_mode`
Returns the current operating mode

Values {'Aborted': '6', 'Modulation': '4', 'On/Off Modulation': '3', 'Waiting for key': '1', 'Continuous': '2', 'Fault': '5', 'Off': '0'}

`mod_mode_changed`

`operating_hours`
Get Laser Head operating hours

`operating_hours_changed`

`power`
Read output power

Units mW

power_changed

power_sp
To handle output power set point (mW) in constant power mode

Units mW

power_sp_changed

restart = functools.partial(<bound method Action.call of <lantz.action.Action object at 0x7f2e5b40c9b0>>, None)

restart_changed

status
Get operating fault

Values {'Constant power time out': '4', 'No errors': '0', 'Temperature error': '1', 'Interlock error': '3'}

status_changed

4.6.5 lantz.drivers.coherent

company Coherent Inc.

description Lasers and Lasers Systems.

website http://www.coherent.com/

---

copyright 2015 by Lantz Authors, see AUTHORS for more details.

license BSD, see LICENSE for more details.

class `lantz.drivers.coherent.`**`Innova300C`**(*resource_name*, *name=None*, ***kwargs*)
Bases: *`lantz.messagebased.MessageBasedDriver`*

Innova300 C Series.

Parameters

- **`resource_name`** (*`str`*) – The resource name
- **`kwargs`** – keyword arguments passed to the resource during initialization.

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

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

`initialize`()

`query`(*command*, *, *send_args=(None, None)*, *recv_args=(None, None)*)
Send query to the laser and return the answer, after handling possible errors.

Parameters **`command`** (*`string`*) – command to be sent to the instrument

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

DEFAULTS = {'ASRL': {'read_termination': '\r\n', 'parity': <Parity.none: 0>, 'baud_rate': 1200, 'bytesize': 8, 'stop_bits

DEFAULTS_changed

analog_enabled

Values {False: 0, True: 1}

analog_enabled_changed

analog_relative

Values {False: 0, True: 1}

analog_relative_changed

auto_light_cal_enabled

Values {False: 0, True: 1}

auto_light_cal_enabled_changed

autofill_delta

Units V

autofill_delta_changed

autofill_mode

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

autofill_mode_changed

autofill_needed

Values {False: 0, True: 1}

autofill_needed_changed

baudrate

RS-232/422 baud rate, the serial connection will be reset after.

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

baudrate_changed

cathode_current

Units A

cathode_current_changed

cathode_voltage

Units V

cathode_voltage_changed

center_powertrack = functools.partial(<bound method Action.call of <lantz.action.Action object at 0x7f2e5b334390>

center_powertrack_changed

control_pin_high

Values {False: 0, True: 1}

control_pin_high_changed

current

Units A

current_change_limit

Limits (5, 100, 1)

`current_change_limit_changed`

`current_changed`

`current_range`

Units A

Limits (10, 100, 1)

`current_range_changed`

`current_setpoint`
Current setpoint when using the current regulation mode.

Units A

Limits (0, 50, 0.01)

`current_setpoint_changed`

`echo_enabled`

Values {False: 0, True: 1}

`echo_enabled_changed`

`etalon_mode`

Values {‘modetune’: 2, ‘manual’: 0, ‘modetrack’: 1}

`etalon_mode_changed`

`etalon_temperature`

Units degC

`etalon_temperature_changed`

`etalon_temperature_setpoint`
Setpoint for the etalon temperature.

Units degC

Limits (51.5, 54, 0.001)

`etalon_temperature_setpoint_changed`

`faults`
List of all active faults.

`faults_changed`

`head_software_rev`

`head_software_rev_changed`

`idn`

`idn_changed`

`initialize_changed`

`is_in_start_delay`
Laser is in start delay (tube not ionized)

`is_in_start_delay_changed`

`laser_enabled`

Values {False: 0, True: 2}

`laser_enabled_changed`

`magnet_current`

Units A

`magnet_current_changed`

`magnetic_field_high`

Values {False: 0, True: 1}

`magnetic_field_high_changed`

`magnetic_field_setpoint_high`
Setpoint for magnetic field setting.

Values {False: 0, True: 1}

`magnetic_field_setpoint_high_changed`

`operating_mode`

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

`operating_mode_changed`

`output_pin_high`

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

`output_pin_high_changed`

`power`

Units A

`power_changed`

`power_setpoint`
Setpoint for the light regulation.

Units W

Limits (0, 50, 0.0001)

`power_setpoint_changed`

`powertrack_mode_enabled`

Values {False: 0, True: 1}

`powertrack_mode_enabled_changed`

`powertrack_position`

Keys ('A', 'B')

Relative position of the PowerTrack solenoids.

Limits (0, 255)

`powertrack_position_changed`

`query_changed`

recalibrate_powertrack = functools.partial(<bound method Action.call of <lantz.action.Action object at 0x7f2e5b3

recalibrate_powertrack_changed

remaining_time

Units hour

remaining_time_changed

software_rev

software_rev_changed

time_to_start

Units second

time_to_start_changed

tube_time

Units hour

tube_time_changed

tube_voltage

Units V

tube_voltage_changed

water_flow

Units gallons/minute

water_flow_changed

water_resistivity

Units kohm*cm

water_resistivity_changed

water_temperature

water_temperature_changed

class lantz.drivers.coherent.**ArgonInnova300C**(*resource_name*, *name=None*, ***kwargs*)
Bases: lantz.drivers.coherent.innova.Innova300C

Argon Innova 300C.

Parameters

- **resource_name** (*str*) – The resource name
- **kwargs** – keyword arguments passed to the resource during initialization.

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

wavelength

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

wavelength_changed

class `lantz.drivers.coherent.`**`KryptonInnova300C`**(*resource_name*, *name=None*, ***kwargs*)
Bases: `lantz.drivers.coherent.innova.Innova300C`

Krypton Innova 300C.

Parameters

- **`resource_name`** (*`str`*) – The resource name
- **`kwargs`** – keyword arguments passed to the resource during initialization.

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

`wavelength`

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

`wavelength_changed`

4.6.6 lantz.drivers.examples

company Lantz Examples.

description Example drivers for simulated instruments.

website

---

copyright 2015 by Lantz Authors, see AUTHORS for more details.

license BSD, see LICENSE for more details.

class `lantz.drivers.examples.`**`LantzSignalGenerator`**(*resource_name*, *name=None*, ***kwargs*)
Bases: *`lantz.messagebased.MessageBasedDriver`*

Lantz Signal Generator

Parameters

- **`resource_name`** (*`str`*) – The resource name
- **`kwargs`** – keyword arguments passed to the resource during initialization.

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

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

`query`(*command*, ***, *send_args=(None*, *None)*, *recv_args=(None*, *None)*)

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

`DEFAULTS` **= {‘COMMON’: {‘read_termination’: ‘\n’, ‘write_termination’: ‘\n’}}**

`DEFAULTS_changed`

`amplitude`
Amplitude.

Units V

Limits (10,)

`amplitude_changed`

`calibrate` = functools.partial(<bound method Action.call of <lantz.action.Action object at 0x7f2e5b250668>>, None)

`calibrate_changed`

`din`

> **Keys** [1, 2, 3, 4, 5, 6, 7, 8]

Digital input state.

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

`din_changed`

`dout`

> **Keys** [1, 2, 3, 4, 5, 6, 7, 8]

Digital output state.

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

`dout_changed`

`frequency`

Frequency.

> **Units** Hz
>
> **Limits** (1, 100000.0)

`frequency_changed`

`idn`

`idn_changed`

`offset`

Offset.

> **Units** V
>
> **Limits** (-5, 5, 0.01)

`offset_changed`

`output_enabled`

Analog output enabled.

> **Values** {False: 0, True: 1}

`output_enabled_changed`

`query_changed`

`self_test` = functools.partial(<bound method Action.call of <lantz.action.Action object at 0x7f2e5b2506d8>>, None)

`self_test_changed`

`waveform`

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

`waveform_changed`

class `lantz.drivers.examples.`**`LantzVoltmeter`**(*resource_name*, *name=None*, ***kwargs*)

Bases: *`lantz.messagebased.MessageBasedDriver`*

Lantz Signal Generator

Parameters

- **`resource_name`** (*`str`*) – The resource name
- **`kwargs`** – keyword arguments passed to the resource during initialization.

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

`auto_range_async`(**args*, ***kwargs*)

(Async) Autoselect a range.

`calibrate_async`(**args*, ***kwargs*)

(Async) Calibrate.

`query`(*command*, ***, *send_args=(None*, *None)*, *recv_args=(None*, *None)*)

`self_test_async`(**args*, ***kwargs*)

(Async) Self test

`DEFAULTS` **= {'COMMON': {'read_termination': '\n', 'write_termination': '\n'}}**

`DEFAULTS_changed`

`auto_range` **= functools.partial(<bound method Action.call of <lantz.action.Action object at 0x7f2e5b21e358>>, None)**

`auto_range_changed`

`calibrate` **= functools.partial(<bound method Action.call of <lantz.action.Action object at 0x7f2e5b21e3c8>>, None)**

`calibrate_changed`

`idn`

`idn_changed`

`query_changed`

`range`

Keys ANY

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

`range_changed`

`self_test` **= functools.partial(<bound method Action.call of <lantz.action.Action object at 0x7f2e5b21e518>>, None)**

`self_test_changed`

`voltage`

Keys (0, 1)

Measure the voltage.

Units V

`voltage_changed`

4.6.7 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`**(*resource_name*, *name=None*, ***kwargs*)
Bases: *`lantz.messagebased.MessageBasedDriver`*

Kentech High Repetition Rate Image Intensifier.

Parameters

- **`resource_name`** (*`str`*) – The resource name
- **`kwargs`** – keyword arguments passed to the resource during initialization.

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

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

`query`(*command*, ***, *send_args=(None*, *None)*, *recv_args=(None*, *None)*)
Send query to the instrument and return the answer. Set remote mode if needed.

`query_expect`(*command*, *read_termination=None*, *expected='ok'*)
Send a query and check that the answer contains the string.

`DEFAULTS` = {'COMMON': {'read_termination': '\n', 'write_termination': '\r'}}

`DEFAULTS_changed`

`average_voltage`
Cathode potential bias with respect of MCP.

Units volt

Limits (-50, 50)

`average_voltage_changed`

`clamp_voltage`
Most negative value of the gate pulse.

Units volt

Limits (-50, 50)

`clamp_voltage_changed`

`clear` = functools.partial(<bound method Action.call of <lantz.action.Action object at 0x7f2e5b1d7550>>, None)

`clear_changed`

`enabled`
MCP Enabled

Values {False, True}

enabled_changed

mcp
MCP Voltage.

Units volt

Limits (0, 1700)

mcp_changed

mode
Gain modulation mode.

HRI Machine Modes and Mode Indices None Mode 0 INHIBIT 2-10 COMB modes 200 ps to 1 ns inclusive (High rate operation) 11-20 COMB modes 100 ps to 3 ns inclusive (Low rate (+GOI) operation) 21 RF 22 logic low duty cycle (LDC) 23 logic high duty cycle 24 DC 25-28 user modes 1 to 4

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

mode_changed

query_changed

query_expect_changed

remote
Remote or local.

Values {False, True}

remote_changed

revision
Revision.

revision_changed

rfgain
RF Gain.

rfgain_changed

status
Get status.

status_changed

temperature
Temperature.

temperature_changed

trigger_ecl_level
Trigger level for ECL logic, mode level.

Units centivolt

Limits (-40, 40, 1)

trigger_ecl_level_changed

trigger_ecl_mode
Trigger mode for ECL logic.

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

trigger_ecl_mode_changed

trigger_edge
Trigger on rising or falling edge.

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

trigger_edge_changed

trigger_logic
Trigger logic.

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

trigger_logic_changed

trigger_ttl_termination
Trigger termination for TTL logic (for ECL is fixed to 50 ohm).

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

trigger_ttl_termination_changed

4.6.8 lantz.drivers.labjack

company LabJack

description LabJacks are USB/Ethernet based measurement and automation devices which provide analog inputs/outputs, digital inputs/outputs, and more.

website http://www.labjack.com/

---

copyright 2015 by Lantz Authors, see AUTHORS for more details.

license BSD, see LICENSE for more details.

class `lantz.drivers.labjack.`**`U12`**(*board_id*)
Bases: `lantz.driver.Driver`

Driver for the Labjack U12 data acquisition device. http://labjack.com/support/u12/users-guide For details about the commands, refer to the users guide.

finalize()

initialize()

analog_dif_in

Keys [0, 1, 2, 3]

Differential channels can make use of the low noise precision PGA to provide gains up to 20. In differential mode, the voltage of each AI with respect to ground must be between +20 and -10 volts, but the range of voltage difference between the 2 AI is a function of gain (G) as follows: G=1 ±20 volts G=2 ±10 volts G=4 ±5 volts G=5 ±4 volts G=8 ±2.5 volts G=10 ±2 volts G=16 ±1.25 volts G=20 ±1 volt The reason the range is ±20 volts at G=1 is that, for example, AI0 could be +10 volts and AI1 could be -10 volts giving a difference of +20 volts, or AI0 could be -10 volts and AI1 could be +10 volts giving a difference of -20 volts. The PGA (programmable gain amplifier, available on differential channels only) amplifies the AI voltage before it is digitized by the A/D converter. The high level drivers then divide the reading by the gain and return the actual measured voltage.

Units volts

`analog_dif_in_changed`

`analog_in`

Keys ANY

Units volts

`analog_in_changed`

`analog_out`

Keys [0, 1]

Easy function. This is a simplified version of AOUpdate. Sets the voltage of both analog outputs. Execution time for this function is 20 milliseconds or less (typically 16 milliseconds in Windows). If either passed voltage is less than zero, the DLL uses the last set voltage. This provides a way to update 1 output without changing the other.

Units volts

`analog_out_changed`

`digital_in_out`

Keys [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]

Easy function. This is a simplified version of DigitalIO that reads the state of one digital input. Also configures the requested pin to input and leaves it that way. Execution time for this function is 20 milliseconds or less (typically 16 milliseconds in Windows). channel – Line to read. 0-3 for IO or 0-15 for D.

Values {False: 0, True: 1}

`digital_in_out_changed`

`finalize_changed`

`initialize_changed`

4.6.9 lantz.drivers.legacy

Legacy drivers based on old instrument classes. Do not use for new code.

copyright 2015 by Lantz Authors, see AUTHORS for more details.

license BSD, see LICENSE for more details.

4.6.10 lantz.drivers.mpb

company MPB Communications Inc.

description Laser products.

website http://www.mpbc.ca/

---

copyright 2015 by Lantz Authors, see AUTHORS for more details.

license BSD, see LICENSE for more details.

class `lantz.drivers.mpb.`**`VFL`**(*resource_name*, *name=None*, ***kwargs*)

Bases: `lantz.messagebased.MessageBasedDriver`

Driver for any VFL MPB Communications laser.

Parameters

- **`resource_name`** (`str`) – The resource name
- **`kwargs`** – keyword arguments passed to the resource during initialization.

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

`tune_shg_async`(**args*, ***kwargs*)

`tune_shg_stop_async`(**args*, ***kwargs*)

`DEFAULTS` **= {'ASRL': {'read_termination': '\r\n', 'parity': <Parity.none: 0>, 'baud_rate': 1200, 'bytesize': 8, 'stop_bits**

`DEFAULTS_changed`

`ctl_mode`

To handle laser diode current (mA) in Active Current Control Mode

Values {'ACC': '0', 'APC': '1'}

`ctl_mode_changed`

`current_sp`

To handle laser diode current (mA) in Active Current Control Mode

Units mA

`current_sp_changed`

`enabled`

Method for turning on the laser

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

`enabled_changed`

`idn`

Identification of the device

`idn_changed`

`ld_current`

To get the laser diode current (mA)

Units mA

`ld_current_changed`

`ld_temp`

To get the laser diode temperature (ºC)

Units degC

`ld_temp_changed`

`power`

To get the laser emission power (mW)

Units mW

`power_changed`

power_sp
To handle output power set point (mW) in APC Mode

Units mW

power_sp_changed

shg_temp
To get the SHG temperature

Units degC

shg_temp_changed

shg_temp_sp
To handle the SHG temperature set point

Units degC

shg_temp_sp_changed

shg_tune_info
Getting information about laser ready for SHG tuning

shg_tune_info_changed

shg_tuning
Initiating SHG tuning

shg_tuning_changed

status
Current device status

status_changed

tec_current
To get the thermoelectric cooler (TEC) current (mA)

Units mA

tec_current_changed

tec_temp
To get the thermoelectric cooler (TEC) temperature (ºC)

Units degC

tec_temp_changed

tune_shg = functools.partial(<bound method Action.call of <lantz.action.Action object at 0x7f2e5b0a67b8>>, None)

tune_shg_changed

tune_shg_stop = functools.partial(<bound method Action.call of <lantz.action.Action object at 0x7f2e5b0a6828>>, No

tune_shg_stop_changed

4.6.11 lantz.drivers.newport

company Newport.

description Test and Measurement Equipment.

website http://www.newport.com/

—

copyright 2015 by Lantz Authors, see AUTHORS for more details.

license BSD,

class `lantz.drivers.newport.`**`PowerMeter1830c`**(*resource_name*, *name=None*, ***kwargs*)
Bases: *`lantz.messagebased.MessageBasedDriver`*

Newport 1830c Power Meter

Parameters

- **`resource_name`** (*`str`*) – The resource name
- **`kwargs`** – keyword arguments passed to the resource during initialization.

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

`autocalibration_async`(**args*, ***kwargs*)
(Async) Autocalibration of the power meter. This procedure disconnects the input signal. It should be performed at least 60 minutes after warm-up.

`store_reference_async`(**args*, ***kwargs*)
(Async) Sets the current input signal power level as the power reference level. Each time the S command is sent, the current input signal becomes the new reference level.

`DEFAULTS` **= {'ASRL': {'read_termination': '\n', 'parity': <Parity.none: 0>, 'baud_rate': 9600, 'bytesize': 8, 'stop_bits':**

`DEFAULTS_changed`

`attenuator`

Attenuator. 1: Attenuator present 0: Attenuator not present

Values {False: 0, True: 1}

`attenuator_changed`

`autocalibration` **= functools.partial(<bound method Action.call of <lantz.action.Action object at 0x7f2e5b093470>>,**

`autocalibration_changed`

`beeper`
Checks whether the audio output is on or off.

Values {False: 0, True: 1}

`beeper_changed`

`data`

`data_changed`

`echo`
Returns echo mode. Only applied to RS232 communication

Values {False: 0, True: 1}

`echo_changed`

`filter`

How many measurements are averaged for the displayed reading. slow: 16 measurements medium: 4 measurements fast: 1 measurement.

Values {'Slow': 1, 'Fast': 3, 'Medium': 2}

`filter_changed`

`go`
Enable or disable the power meter from taking new measurements.

Values {False: 0, True: 1}

`go_changed`

`keypad`
Keypad/Display backlight intensity levels.

Values {‘High’: 2, ‘Off’: 0, ‘Medium’: 1}

`keypad_changed`

`lockout`
Enable/Disable the lockout. When the lockout is enabled, any front panel key presses would have no effect on system operation.

Values {False: 0, True: 1}

`lockout_changed`

`range`

Set the signal range for the input signal. 0 means auto setting the range. 1 is the lowest signal range and 8 the highest.

Values {0, 1, 2, 3, 4, 5, 6, 7, 8}

`range_changed`

`store_reference` = functools.partial(<bound method Action.call of <lantz.action.Action object at 0x7f2e5b093780>>,

`store_reference_changed`

`units`
Sets and gets the units of the measurements.

Values {‘dB’: 2, ‘REL’: 4, ‘Watts’: 1, ‘dBm’: 3}

`units_changed`

`wavelength`
Sets and gets the wavelength of the input signal.

Limits (1, 10000, 1)

`wavelength_changed`

`zero`
Turn the zero function on/off. Zero function is used for subtracting any background power levels in future measurements.

Values {False: 0, True: 1}

`zero_changed`

4.6.12 lantz.drivers.ni

company National Instruments

description

website http://www.ni.com/

4.6.13 lantz.drivers.olympus

company Olympus.

description Research and clinical microscopes.

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

—

class `lantz.drivers.olympus.`**`IX2`**(*resource_name*, *name=None*, ***kwargs*)

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

Olympus IX2 Body

Parameters

- **`resource_name`** (*`str`*) – The resource name
- **`kwargs`** – keyword arguments passed to the resource during initialization.

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

`bottom_port_closed`

Values {False: ‘OUT’, True: ‘IN’}

`bottom_port_closed_changed`

`camera_port_enabled`

Values {False: ‘2’, True: ‘1’}

`camera_port_enabled_changed`

`condensor`

Get procs

- <class ‘int’>:set procs: - <class ‘str’>

`condensor_changed`

`filter_wheel`

Get procs

- <class ‘int’>:set procs: - <class ‘str’>

`filter_wheel_changed`

`mirror_unit`

Get procs

- <class ‘int’>:set procs: - <class ‘str’>

mirror_unit_changed

shutter1_closed

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

shutter1_closed_changed

shutter2_closed

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

shutter2_closed_changed

class `lantz.drivers.olympus.`**BX2A**(*resource_name*, *name=None*, ***kwargs*)

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

Olympus BX2A Body

Parameters

- **resource_name** (*str*) – The resource name
- **kwargs** – keyword arguments passed to the resource during initialization.

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

aperture_stop_diameter

Get procs

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

aperture_stop_diameter_changed

condenser_top_lens_enabled

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

condenser_top_lens_enabled_changed

configure_filterwheel

Get procs

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

configure_filterwheel_changed

cube

Get procs

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

cube_changed

shutter_closed

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

shutter_closed_changed

turret

Get procs

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

turret_changed

class `lantz.drivers.olympus.`**`IXBX`**(*resource_name*, *name=None*, ***kwargs*)
Bases: *`lantz.messagebased.MessageBasedDriver`*

IX or BX Olympus microscope body.

Parameters

- **`resource_name`** (*`str`*) – The resource name
- **`kwargs`** – keyword arguments passed to the resource during initialization.

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

`init_origin`()
Init origin

`initialize`()

`lamp_status`()

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

`query`(*command*, ***, *send_args=(None*, *None)*, *recv_args=(None*, *None)*)
Query the instrument and parse the response.

Raises InstrumentError

`stop`()
Stop any currently executing motion

`DEFAULTS` **= {'ASRL': {'read_termination': '\r\n', 'parity': <Parity.even: 2>, 'baud_rate': 19200, 'bytesize': 8, 'stop_bit**

`DEFAULTS_changed`

`body_locked`

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

`body_locked_changed`

`fluo_shutter`

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

`fluo_shutter_changed`

`focus_locked`

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

`focus_locked_changed`

`idn`
Microscope identification

`idn_changed`

`init_origin_changed`

`initialize_changed`

`jog_dial`

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

`jog_dial_changed`

`jog_enabled`

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

`jog_enabled_changed`

`jog_limit_enabled`

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

`jog_limit_enabled_changed`

`jog_sensitivity`

Get procs

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

`jog_sensitivity_changed`

`lamp_enabled`

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

`lamp_enabled_changed`

`lamp_epi_enabled`

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

`lamp_epi_enabled_changed`

`lamp_intensity`

Get procs

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

`lamp_intensity_changed`

`lamp_status_changed`

`move_relative` = functools.partial(<bound method Action.call of <lantz.action.Action object at 0x7f2e5ab791d0>>, No

`move_relative_changed`

`move_to_start_enabled`

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

`move_to_start_enabled_changed`

`movement_status`

`movement_status_changed`

`objective`

Get procs

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

`objective_changed`

`query_changed`

`soft_limits`

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

`soft_limits_changed`

`stop_changed`

z
Position of the objective.

Units 0.01 micrometer

z_changed

4.6.14 lantz.drivers.pco

company PCO.

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

website http://www.pco.de

—

copyright 2015 by Lantz Authors, see AUTHORS for more details.

license BSD, see LICENSE for more details.

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’**

`LIBRARY_NAME_changed`

`bel_time`

Units microseconds

`bel_time_changed`

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

`binning_changed`

`coc`
Command Operation Code

`coc_changed`

`coc_time`

Units microseconds

`coc_time_changed`

`delay_time`

Units microseconds

`delay_time_changed`

`exp_time`

Units microseconds

`exp_time_changed`

`expose` = functools.partial(<bound method Action.call of <lantz.action.Action object at 0x7f2e5aa64e80>>, None)

`expose_changed`

`exposure_time`
Exposure time.

Units ms

`exposure_time_changed`

`finalize_changed`

`image_size`
Image size in pixels (width, height).

`image_size_changed`

`image_status`
Image status

`image_status_changed`

`initialize_changed`

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

`mode_changed`

`read_out` = functools.partial(<bound method Action.call of <lantz.action.Action object at 0x7f2e5aa64d68>>, None)

`read_out_changed`

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

`roi_changed`

`run_coc` = functools.partial(<bound method Action.call of <lantz.action.Action object at 0x7f2e5aa69160>>, None)

`run_coc_changed`

`status`

`status_changed`

`stop_coc` = functools.partial(<bound method Action.call of <lantz.action.Action object at 0x7f2e5aa69240>>, None)

`stop_coc_changed`

`table`
COC table

`table_changed`

`take_image` = functools.partial(<bound method Action.call of <lantz.action.Action object at 0x7f2e5aa64c50>>, None)

`take_image_changed`

`trigger`
Triger mode.

`trigger_changed`

4.6.15 lantz.drivers.prior

company PRIOR Scientific

description Microscope Automation & Custom Solutions

website http://www.prior.com/

---

copyright 2015 by Lantz Authors, see AUTHORS for more details.

license BSD, see LICENSE for more details.

class `lantz.drivers.prior.`**`NanoScanZ`**(*resource_name*, *name=None*, ***kwargs*)
Bases: *`lantz.messagebased.MessageBasedDriver`*

Driver for the NanoScanZ Nano Focusing Piezo Stage from Prior.

Parameters

- **`resource_name`** (*`str`*) – The resource name
- **`kwargs`** – keyword arguments passed to the resource during initialization.

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

`move_absolute_async`(**args*, ***kwargs*)
(Async) Move to absolute position n, range (0,100). This is a "real" absolute position and is independent of any relative offset added by the position Feat.

`move_relative_async`(**args*, ***kwargs*)
(Async) Move the stage position relative to the current position by an amount determined by 'value'. If value is given in micrometer, thats the amount the stage is going to move, in microns. If value is given in steps, the stage will move a distance value.magnitude * step. The step is defined by the step Feat

`query`(*command*, ***, *send_args=(None, None)*, *recv_args=(None, None)*)
Send query to the stage and return the answer, after handling possible errors.

Parameters **`command`** (*`string`*) – command to be sent to the instrument

`zero_position_async`(**args*, ***kwargs*)
(Async) Move to zero including any position redefinitions done by the position Feat

`DEFAULTS` = {'ASRL': {'read_termination': '\n', 'parity': <Parity.none: 0>, 'baud_rate': 9600, 'bytesize': 8, 'stop_bits':

`DEFAULTS_changed`

device_baudrate

Reports and sets the baud rate. NOTE: DO NOT change the baud rate of the Piezo controller when daisy chained to ProScan.

Values {38400, 19200, 9600}

device_baudrate_changed

idn
Identification of the device

idn_changed

move_absolute = functools.partial(<bound method Action.call of <lantz.action.Action object at 0x7f2e5a9fd4e0>>, No

move_absolute_changed

move_relative = functools.partial(<bound method Action.call of <lantz.action.Action object at 0x7f2e5a9fd630>>, No

move_relative_changed

moving
Returns the movement status, 0 stationary, 4 moving

Values {False: ‘00000’, True: ‘4’}

moving_changed

position

Gets and sets current position. If the value is set to z = 0, the display changes to REL 0 (relative display mode). To return to ABS mode use inst.move_absolute(0) and then inst.position = 0. Thus, the stage will return to 0 micrometers and the display screen will switch to ABS mode.

Units micrometer

position_changed

query_changed

software_version
Software version

software_version_changed

step
Report and set the default step size, in microns

Units micrometer

step_changed

zero_position = functools.partial(<bound method Action.call of <lantz.action.Action object at 0x7f2e5a9fd470>>, No

zero_position_changed

4.6.16 lantz.drivers.rgblasersystems

company RGB Lasersysteme GmbH.

description Lasers and Lasers Systems.

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

copyright 2015 by Lantz Authors, see AUTHORS for more details.

license BSD, see LICENSE for more details.

class `lantz.drivers.rgblasersystems.`**`MiniLasEvo`**(*resource_name*, *name=None*, ***kwargs*)
Bases: *`lantz.messagebased.MessageBasedDriver`*

Driver for any RGB Lasersystems MiniLas Evo laser.

Parameters

- **`resource_name`** (*`str`*) – The resource name
- **`kwargs`** – keyword arguments passed to the resource during initialization.

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

`initialize`()

`query`(*command*, ***, *send_args=(None*, *None)*, *recv_args=(None*, *None)*)
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

`DEFAULTS` = **{'ASRL': {'read_termination': '\r\n', 'parity': <Parity.none: 0>, 'baud_rate': 57600, 'bytesize': 8, 'stop_bit**

`DEFAULTS_changed`

`available_features`
Available features (reserved for future use)

`available_features_changed`

`control_mode`
Active current (power) control

`control_mode_changed`

`emission_wavelenght`
Emission wavelenght in nm

`emission_wavelenght_changed`

`enabled`
Method for turning on the laser

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

`enabled_changed`

`idn`
Identification of the device

`idn_changed`

`initialize_changed`

`maximum_power`
Gets the maximum emission power of the laser

Units mW

`maximum_power_changed`

`operating_hours`
Total operating hours [hhhh:mm]

`operating_hours_changed`

`power`
Gets and sets the emission power

Units mW

`power_changed`

`power_sp`
Gets and sets the emission power

Units mW

`power_sp_changed`

`query_changed`

`software_version`
Software version

`software_version_changed`

`status`
Current device status

`status_changed`

`temperature`
Current temperature in ºC

`temperature_changed`

`temperature_max`
Highest operating temperature in ºC

`temperature_max_changed`

`temperature_min`
Lowest operating temperature in ºC

`temperature_min_changed`

4.6.17 lantz.drivers.rigol

company Rigol.

description Acquisition devices

website http://www.rigolna.com/

---

class `lantz.drivers.rigol.`**`DS1052e`**(*resource_name*, *name=None*, ***kwargs*)
Bases: *`lantz.messagebased.MessageBasedDriver`*

Parameters

- **resource_name** (*str*) – The resource name
- **kwargs** – keyword arguments passed to the resource during initialization.

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

4.6.18 lantz.drivers.stanford

company Standford Research Systems.

description Manufactures test instruments for research and industrial applications

website http://www.thinksrs.com/

---

copyright 2015 by Lantz Authors, see AUTHORS for more details.

license BSD, see LICENSE for more details.

class lantz.drivers.stanford.**SR830**(*resource_name*, *name=None*, ***kwargs*)
Bases: *lantz.messagebased.MessageBasedDriver*

Parameters

- **resource_name** (*str*) – The resource name
- **kwargs** – keyword arguments passed to the resource during initialization.

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

auto_gain_async_async(**args*, ***kwargs*)
(Async) Equivalent to press the Auto Gain key in the front panel. Might take some time if the time constant is long. Does nothing if the constant is greater than 1 second.

auto_offset_async_async(**args*, ***kwargs*)
(Async) Automatically offset a given channel to zero. Is equivalent to press the Auto Offset Key in the front panel.

Parameters **channel_name** – the name of the channel.

auto_phase_async_async(**args*, ***kwargs*)
(Async) Equivalent to press the Auto Phase key in the front panel. Might take some time if the time constant is long. Does nothing if the phase is unstable.

auto_reserve_async_async(**args*, ***kwargs*)
(Async) Equivalent to press the Auto Reserve key in the front panel. Might take some time if the time constant is long.

measure_async(**args*, ***kwargs*)

pause_data_storage_async(**args*, ***kwargs*)
(Async) Pause data storage

read_buffer_async(**args*, ***kwargs*)
(Async) Queries points stored in the Channel buffer

Parameters

- **channel** – Number of the channel (1, 2).
- **start** – Index of the buffer to start.

- **`length`** – Number of points to read. Defaults to the number of points in the buffer.
- **`format`** – Transfer format 'a': ASCII (slow) 'b': IEEE Binary (fast) - NOT IMPLEMENTED 'c': Non-IEEE Binary (fastest) - NOT IMPLEMENTED

`recall_state_async`(**args*, ***kwargs*)
(Async) Recalls instrument state in specified non-volatile location.

Parameters **`location`** – non-volatile storage location.

`reset_data_storage_async`(**args*, ***kwargs*)
(Async) Reset data buffers. The command can be sent at any time - any storage in progress, paused or not. will be reset. The command will erase the data buffer.

`save_state_async`(**args*, ***kwargs*)
(Async) Saves instrument state in specified non-volatile location.

Previously stored state in location is overwritten (no error is generated). :param location: non-volatile storage location.

`start_data_storage_async`(**args*, ***kwargs*)
(Async) Start or resume data storage

`trigger_async`(**args*, ***kwargs*)
(Async) Software trigger.

`wait_bit1`()

`DEFAULTS` **= {'COMMON': {'read_termination': '\n', 'write_termination': '\n'}}**

`DEFAULTS_changed`

`alarm_enabled`
Key click

Values {False: 0, True: 1}

`alarm_enabled_changed`

`analog_input`

Keys {1, 2, 3, 4}

Input voltage in the auxiliary analog input.

Units volt

`analog_input_changed`

`analog_output`

Keys {1, 2, 3, 4}

Ouput voltage in the auxiliary analog output.

Units volt

Limits (-10.5, 10.5, 0.001)

`analog_output_changed`

`analog_value`

Keys ANY

Units volt

`analog_value_changed`

auto_gain = functools.partial(<bound method Action.call of <lantz.action.Action object at 0x7f2e5a905748>>, None)

auto_gain_async = functools.partial(<bound method Action.call of <lantz.action.Action object at 0x7f2e5a9056a0>>,

auto_gain_async_changed

auto_gain_changed

auto_offset = functools.partial(<bound method Action.call of <lantz.action.Action object at 0x7f2e5a905b38>>, None)

auto_offset_async = functools.partial(<bound method Action.call of <lantz.action.Action object at 0x7f2e5a905a90>

auto_offset_async_changed

auto_offset_changed

auto_phase = functools.partial(<bound method Action.call of <lantz.action.Action object at 0x7f2e5a9059e8>>, None)

auto_phase_async = functools.partial(<bound method Action.call of <lantz.action.Action object at 0x7f2e5a905940>>

auto_phase_async_changed

auto_phase_changed

auto_reserve = functools.partial(<bound method Action.call of <lantz.action.Action object at 0x7f2e5a905898>>, Non

auto_reserve_async = functools.partial(<bound method Action.call of <lantz.action.Action object at 0x7f2e5a9057f0

auto_reserve_async_changed

auto_reserve_changed

buffer_length

buffer_length_changed

display

Keys {1, 2}

Front panel output source.

display_changed

filter_db_per_oct

Time constant.

Values {24, 18, 12, 6}

filter_db_per_oct_changed

frequency

Reference frequency.

Units Hz

Limits (0.001, 102000, 1e-05)

frequency_changed

front_output

Keys {1, 2}

Front panel output source.

Values {'xy': 1, 'display': 0}

front_output_changed

harmonic
Detection harmonic.

Limits (1, 19999, 1)

harmonic_changed

input_configuration
Configuration of the Input.

Values {'A-B': 1, 'I100': 3, 'A': 0, 'I1': 2}

input_configuration_changed

input_coupling
Input coupling.

Values {'DC': 1, 'AC': 0}

input_coupling_changed

input_filter
Input line notch filters (1x, 2x).

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

input_filter_changed

input_shield
Input shield grounding.

Values {'ground': 1, 'float': 0}

input_shield_changed

key_click_enabled
Key click

Values {False: 0, True: 1}

key_click_enabled_changed

measure = functools.partial(<bound method Action.call of <lantz.action.Action object at 0x7f2e5a90f7f0>>, None)

measure_changed

pause_data_storage = functools.partial(<bound method Action.call of <lantz.action.Action object at 0x7f2e5a90f390

pause_data_storage_changed

read_buffer = functools.partial(<bound method Action.call of <lantz.action.Action object at 0x7f2e5a90fa20>>, None)

read_buffer_changed

recall_state = functools.partial(<bound method Action.call of <lantz.action.Action object at 0x7f2e5a9054e0>>, Non

recall_state_changed

reference_internal
Reference source.

Values {False: 0, True: 1}

reference_internal_changed

reference_phase_shift
Phase shift of the reference.

Units degrees

Limits (-360, 729.99, 0.01)

reference_phase_shift_changed

reference_trigger
Reference trigger when using the external reference mode.

Values {'zero_crossing': 0, 'rising_edge': 1}

reference_trigger_changed

remote
Lock Front panel.

Values {False: 1, True: 0}

remote_changed

reserve_mode
Reserve mode.

Values {'normal': 1, 'high': 0, 'low': 2}

reserve_mode_changed

reset_data_storage = functools.partial(<bound method Action.call of <lantz.action.Action object at 0x7f2e5a90f518

reset_data_storage_changed

sample_rate
Sample rate.

Values OrderedDict([('62.5 mHz', 0), ('125 mHz', 1), ('250 mHz', 2), ('500 mHz', 3), ('1 Hz', 4), ('2 Hz', 5), ('4 Hz', 6), ('8 Hz', 7), ('16 Hz', 8), ('32 Hz', 9), ('64 Hz', 10), ('128 Hz', 11), ('256 Hz', 12), ('512 Hz', 13), ('trigger', 14)])

sample_rate_changed

save_state = functools.partial(<bound method Action.call of <lantz.action.Action object at 0x7f2e5a905550>>, None)

save_state_changed

sensitivity
Sensitivity.

Values OrderedDict([('2 nV/fA', 0), ('5 nV/fA', 1), ('10 nV/fA', 2), ('20 nV/fA', 3), ('50 nV/fA', 4), ('100 nV/fA', 5), ('200 nV/fA', 6), ('500 nV/fA', 7), ('1 uV/pA', 8), ('2 uV/pA', 9), ('5 uV/pA', 10), ('10 uV/pA', 11), ('20 uV/pA', 12), ('50 uV/pA', 13), ('100 uV/pA', 14), ('200 uV/pA', 15), ('500 uV/pA', 16), ('1 mV/nA', 17), ('2 mV/nA', 18), ('5 mV/nA', 19), ('10 mV/nA', 20), ('20 mV/nA', 21), ('50 mV/nA', 22), ('100 mV/nA', 23), ('200 mV/nA', 24), ('500 mV/nA', 25), ('1 V/uA', 26)])

sensitivity_changed

sine_output_amplitude
Amplitude of the sine output.

Units volt

Limits (0.004, 5.0, 0.002)

sine_output_amplitude_changed

single_shot
End of buffer mode.

If loop mode (single_shot = False), make sure to pause data storage before reading the data to avoid confusion about which point is the most recent.

Values {False: 1, True: 0}

single_shot_changed

start_data_storage = functools.partial(<bound method Action.call of <lantz.action.Action object at 0x7f2e5a90f320

start_data_storage_changed

sync_filter
Synchronous filter status.

Values {False: 0, True: 1}

sync_filter_changed

time_constants
Time constant.

Values OrderedDict([('10 us', 0), ('30 us', 1), ('100 us', 2), ('300 us', 3), ('1 ms', 4), ('3 ms', 5), ('10 ms', 6), ('30 ms', 7), ('100 ms', 8), ('300 ms', 9), ('1 s', 10), ('3 s', 11), ('10 s', 12), ('30 s', 13), ('100 s', 14), ('300 s', 15), ('1 ks', 16), ('3 ks', 17), ('10 ks', 18), ('30 ks', 19)])

time_constants_changed

trigger = functools.partial(<bound method Action.call of <lantz.action.Action object at 0x7f2e5a90f0b8>>, None)

trigger_changed

trigger_start_mode

trigger_start_mode_changed

wait_bit1_changed

4.6.19 lantz.drivers.sutter

company Sutter Instrument.

description Biomedical and scientific instrumentation.

website http://www.sutter.com/

—

copyright 2015 by Lantz Authors, see AUTHORS for more details.

license BSD, see LICENSE for more details.

class lantz.drivers.sutter.**Lambda103**(*resource_name*, *name=None*, ***kwargs*)
Bases: *lantz.messagebased.MessageBasedDriver*

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

Parameters

- **resource_name** (*str*) – The resource name
- **kwargs** – keyword arguments passed to the resource during initialization.

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

flush()
Flush.

initialize()

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

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

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

DEFAULTS = {'ASRL': {'read_termination': '', 'write_termination': ''}}

DEFAULTS_changed

flush_changed

initialize_changed

motorsON = functools.partial(<bound method Action.call of <lantz.action.Action object at 0x7f2e5a7ef940>>, None)

motorsON_changed

open_A
Open shutter A.

Values {False: '¬', True: 'ª'}

open_A_changed

position

Keys {'B': 1, 'A': 0}

Set filter wheel position and speed.

w = 0 -> Filter wheels A and C w = 1 -> Fliter wheel B

position_changed

remote
Set Local-Mode.

Values {False: 'ï', True: 'î'}

remote_changed

reset = functools.partial(<bound method Action.call of <lantz.action.Action object at 0x7f2e5a7efcf8>>, None)

reset_changed

status = functools.partial(<bound method Action.call of <lantz.action.Action object at 0x7f2e5a7ef9b0>>, None)

status_changed

4.6.20 lantz.drivers.tektronix

company Tektronix.

description Test and Measurement Equipment.

website http://www.tek.com/

—

copyright 2015 by Lantz Authors, see AUTHORS for more details.

license BSD,

class `lantz.drivers.tektronix.`**`TDS2024`**(*resource_name*, *name=None*, ***kwargs*)

Bases: *`lantz.messagebased.MessageBasedDriver`*

Tektronix TDS2024 200 MHz 4 Channel Digital Real-Time Oscilloscope

Parameters

- **`resource_name`** (*`str`*) – The resource name
- **`kwargs`** – keyword arguments passed to the resource during initialization.

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

`acqparams_async`(**args*, ***kwargs*)
(Async) X/Y Increment Origin and Offset.

`autoconf_async`(**args*, ***kwargs*)
(Async) Autoconfig oscilloscope.

`curv_async`(**args*, ***kwargs*)
(Async) Get data.

Returns: xdata, data as list

`dataencoding_async`(**args*, ***kwargs*)
(Async) Set data encoding.

`datasource_async`(**args*, ***kwargs*)
(Async) Selects channel.

`forcetrigger_async`(**args*, ***kwargs*)
(Async) Force trigger event.

`initialize`()
initiate.

`measure_frequency_async`(**args*, ***kwargs*)
(Async) Get immediate measurement result.

`measure_max_async`(**args*, ***kwargs*)
(Async) Get immediate measurement result.

`measure_mean_async`(**args*, ***kwargs*)
(Async) Get immediate measurement result.

`measure_min_async`(**args*, ***kwargs*)
(Async) Get immediate measurement result.

`triggerlevel_async`(**args*, ***kwargs*)
(Async) Set trigger level to 50% of the minimum adn maximum values of the signal.

`MANUFACTURER_ID` = '0x699'

`MANUFACTURER_ID_changed`

`acqparams` = functools.partial(<bound method Action.call of <lantz.action.Action object at 0x7f2e5a7b5c88>>, None)

`acqparams_changed`

`autoconf` = functools.partial(<bound method Action.call of <lantz.action.Action object at 0x7f2e5a7b5dd8>>, None)

`autoconf_changed`

`curv` = functools.partial(<bound method Action.call of <lantz.action.Action object at 0x7f2e5a7b5588>>, None)

curv_changed

dataencoding = functools.partial(<bound method Action.call of <lantz.action.Action object at 0x7f2e5a7b55c0>>, Non

dataencoding_changed

datasource = functools.partial(<bound method Action.call of <lantz.action.Action object at 0x7f2e5a7b5d30>>, None)

datasource_changed

forcetrigger = functools.partial(<bound method Action.call of <lantz.action.Action object at 0x7f2e5a7b5668>>, Non

forcetrigger_changed

idn
IDN.

idn_changed

initialize_changed

measure_frequency = functools.partial(<bound method Action.call of <lantz.action.Action object at 0x7f2e5a7b59e8>

measure_frequency_changed

measure_max = functools.partial(<bound method Action.call of <lantz.action.Action object at 0x7f2e5a7b53c8>>, None)

measure_max_changed

measure_mean = functools.partial(<bound method Action.call of <lantz.action.Action object at 0x7f2e5a7b5978>>, Non

measure_mean_changed

measure_min = functools.partial(<bound method Action.call of <lantz.action.Action object at 0x7f2e5a7b5400>>, None)

measure_min_changed

trigger
Trigger state.

trigger_changed

triggerlevel = functools.partial(<bound method Action.call of <lantz.action.Action object at 0x7f2e5a7b72b0>>, Non

triggerlevel_changed

class lantz.drivers.tektronix.**TDS1002b**(*resource_name*, *name=None*, ***kwargs*)
Bases: *lantz.messagebased.MessageBasedDriver*

Parameters

- **resource_name** (*str*) – The resource name
- **kwargs** – keyword arguments passed to the resource during initialization.

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

MANUFACTURER_ID = '0x699'

MANUFACTURER_ID_changed

MODEL_CODE = '0x363'

MODEL_CODE_changed

idn

idn_changed

class `lantz.drivers.tektronix.`**`TDS1012`**(*resource_name*, *name=None*, ***kwargs*)
Bases: `lantz.messagebased.MessageBasedDriver`

Tektronix TDS1012 100MHz 2 Channel Digital Storage Oscilloscope

Parameters

- **`resource_name`** (`str`) – The resource name
- **`kwargs`** – keyword arguments passed to the resource during initialization.

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

`acquire_curve_async`(**args*, ***kwargs*)
(Async) Gets data from the oscilloscope. It accepts setting the start and stop points of the acquisition (by default the entire range).

`acquire_parameters_async`(**args*, ***kwargs*)
(Async) Acquire parameters of the osciloscope. It is intended for adjusting the values obtained in acquire_curve

`autocal_async`(**args*, ***kwargs*)
(Async) Autocalibration of osciloscope. It may take several minutes to complete

`autoset_async`(**args*, ***kwargs*)
(Async) Adjust the vertical, horizontal and trigger controls to display a stable waveform.

`data_setup_async`(**args*, ***kwargs*)
(Async) Sets the way data is going to be encoded for sending.

`forcetrigger_async`(**args*, ***kwargs*)
(Async) Creates a trigger event.

`idn_async`(**args*, ***kwargs*)
(Async) Identify the Osciloscope

`initiate_async`(**args*, ***kwargs*)
(Async) Initiates the acquisition in the osciloscope.

`measure_frequency`()
Gets the frequency of the signal.

`measure_maximum`()
Gets the mean of the signal.

`measure_mean`()
Gets the mean of the signal.

`measure_minimum`()
Gets the minimum of the signal.

`triggerlevel_async`(**args*, ***kwargs*)
(Async) Sets the trigger level to 50% of the minimum and maximum values of the signal.

`MANUFACTURER_ID` = **'0x699'**

`MANUFACTURER_ID_changed`

`acquire_curve` = **functools.partial(<bound method Action.call of <lantz.action.Action object at 0x7f2e5a798dd8>>, No**

`acquire_curve_changed`

`acquire_parameters` = **functools.partial(<bound method Action.call of <lantz.action.Action object at 0x7f2e5a798be0**

`acquire_parameters_changed`

`autocal` = **functools.partial(<bound method Action.call of <lantz.action.Action object at 0x7f2e5a7989b0>>, None)**

`autocal_changed`

`autoset` = **functools.partial(<bound method Action.call of <lantz.action.Action object at 0x7f2e5a798908>>, None)**

`autoset_changed`

`data_setup` = **functools.partial(<bound method Action.call of <lantz.action.Action object at 0x7f2e5a798c50>>, None)**

`data_setup_changed`

`datasource`

Retrieves the data source from which data is going to be taken. TDS1012 has 2 channels

Limits (1, 2)

`datasource_changed`

`forcetrigger` = **functools.partial(<bound method Action.call of <lantz.action.Action object at 0x7f2e5a798e48>>, Non**

`forcetrigger_changed`

`horizontal_division`
Horizontal time base division.

`horizontal_division_changed`

`idn` = **functools.partial(<bound method Action.call of <lantz.action.Action object at 0x7f2e5a798780>>, None)**

`idn_changed`

`initiate` = **functools.partial(<bound method Action.call of <lantz.action.Action object at 0x7f2e5a798710>>, None)**

`initiate_changed`

`measure_frequency_changed`

`measure_maximum_changed`

`measure_mean_changed`

`measure_minimum_changed`

`number_averages`
Number of averages

Values {128, 64, 16, 0, 4}

`number_averages_changed`

`trigger`
Retrieves trigger state.

Values {‘NORMAL’, ‘AUTO’}

`trigger_changed`

`triggerlevel` = **functools.partial(<bound method Action.call of <lantz.action.Action object at 0x7f2e5a798ef0>>, None**

`triggerlevel_changed`

AA Opto Electronic.

- *MDSnC synthesizer for AOTF.nC*

Aeroflex

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

Andor

- *Andor*
- *CCD*
- *Neo Andor CMOS Camera*

Cobolt.

- *Driver for any Cobolt 06-01 Series laser.*

Coherent Inc.

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

Lantz Examples.

- *Lantz Signal Generator*
- *Lantz Signal Generator*

Kentech Instruments Ltd.

- *Kentech High Repetition Rate Image Intensifier.*

LabJack

- :class:‘ <lantz.drivers.labjack.U12>‘

drivers

MPB Communications Inc.

- *Driver for any VFL MPB Communications laser.*

Newport.

- *Newport 1830c Power Meter*

National Instruments

Olympus.

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

PCO.

- *PCO Sensicam*

PRIOR Scientific

- *Driver for the NanoScanZ Nano Focusing Piezo Stage from Prior.*

RGB Lasersysteme GmbH.

- *Driver for any RGB Lasersystems MiniLas Evo laser.*

Rigol.

- *DS1052e*

Standford Research Systems.

- *SR830*

Sutter Instrument.

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

Tektronix.

- *TDS1002b*
- *Tektronix TDS1012 100MHz 2 Channel Digital Storage Oscilloscope*
- *Tektronix TDS2024 200 MHz 4 Channel Digital Real-Time Oscilloscope*

4.7 API

4.7.1 General

class `lantz.`**`Driver`**(**args*, ***kw*)

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=None*, *, *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=None*, *, *force=False*, ***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=None*, *, *force=False*, *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=MISSING*, *fset=None*, *doc=None*, *, *values=None*, *units=None*, *limits=None*, *procs=None*, *read_once=False*)
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=MISSING*, *fset=None*, *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=None*, *, *values=None*, *units=None*, *limits=None*, *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

4.7.2 Interfacing to instruments

lantz.messagebased

Implementes base class for message based drivers using PyVISA under the hood.

copyright 2015 by Lantz Authors, see AUTHORS for more details.

license BSD, see LICENSE for more details.

class lantz.messagebased.**MessageBasedDriver** (*resource_name*, *name=None*, ***kwargs*)
Bases: lantz.driver.Driver

Base class for message based drivers using PyVISA as underlying library.

Notice that PyVISA can communicate using different backends. For example: - @ni: Using NI-VISA for communication. Backend bundled with PyVISA. - @py: Using PySerial, PyUSB and linux-gpib for communication. Available with PyVISA-py package. - @sim: Simulated devices. Available with PyVISA-sim package.

Parameters

- **resource_name** (*str*) – The resource name
- **kwargs** – keyword arguments passed to the resource during initialization.

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

parse_query (*command*, *, *send_args=(None*, *None)*, *recv_args=(None*, *None)*, *format=None*)
Send query to the instrument, parse the output using format and return the answer.

See also:

TextualMixin.query and stringparser

query (*command*, *, *send_args=(None*, *None)*, *recv_args=(None*, *None)*)
Send query to the instrument and return the answer

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

read (*termination=None*, *encoding=None*)
Receive string from instrument.

Parameters

- **termination** (*str*) – termination character (overrides class default)
- **encoding** – encoding to transform bytes to string (overrides class default)

Returns string encoded from received bytes

classmethod via_gpib (*address*, *name=None*, ***kwargs*)
Return a Driver with an underlying GPIB Instrument resource.

Parameters

- **address** – The gpib address of the instrument.
- **name** – Unique name given within Lantz to the instrument for logging purposes. Defaults to one generated based on the class name if not provided.
- **kwargs** – keyword arguments passed to the Resource constructor on initialize.

Return type *MessageBasedDriver*

classmethod via_serial (*port*, *name=None*, ***kwargs*)
Return a Driver with an underlying ASRL (Serial) Instrument resource.

Parameters

- **port** – The serial port to which the instrument is connected.
- **name** – Unique name given within Lantz to the instrument for logging purposes. Defaults to one generated based on the class name if not provided.
- **kwargs** – keyword arguments passed to the Resource constructor on initialize.

Return type *MessageBasedDriver*

classmethod via_tcpip (*hostname*, *port*, *name=None*, ***kwargs*)
Return a Driver with an underlying TCP Instrument resource.

Parameters

- **hostname** – The ip address or hostname of the instrument.

- **port** – the port of the instrument.
- **name** – Unique name given within Lantz to the instrument for logging purposes. Defaults to one generated based on the class name if not provided.
- **kwargs** – keyword arguments passed to the Resource constructor on initialize.

Return type *MessageBasedDriver*

classmethod via_tcpip_socket(*hostname*, *port*, *name=None*, ***kwargs*)

Return a Driver with an underlying TCP Socket resource.

Parameters

- **hostname** – The ip address or hostname of the instrument.
- **port** – the port of the instrument.
- **name** – Unique name given within Lantz to the instrument for logging purposes. Defaults to one generated based on the class name if not provided.
- **kwargs** – keyword arguments passed to the Resource constructor on initialize.

Return type *MessageBasedDriver*

classmethod via_usb(*serial_number=None*, *manufacturer_id=None*, *model_code=None*, *name=None*, *board=0*, ***kwargs*)

Return a Driver with an underlying USB Instrument resource.

A connected USBTMC instrument with the specified serial_number, manufacturer_id, and model_code is returned. If any of these is missing, the first USBTMC driver matching any of the provided values is returned.

To specify the manufacturer id and/or the model code override the following class attributes:

```
class RigolDS1052E(MessageBasedDriver):

    MANUFACTURER_ID = '0x1AB1'
    MODEL_CODE = '0x0588'
```

Parameters

- **serial_number** – The serial number of the instrument.
- **manufacturer_id** – The unique identification number of the manufacturer.
- **model_code** – The unique identification number of the product.
- **name** – Unique name given within Lantz to the instrument for logging purposes. Defaults to one generated based on the class name if not provided.
- **board** – USB Board to use
- **kwargs** – keyword arguments passed to the Resource constructor on initialize.

Return type *MessageBasedDriver*

classmethod via_usb_raw(*serial_number=None*, *manufacturer_id=None*, *model_code=None*, *name=None*, *board=0*, ***kwargs*)

Return a Driver with an underlying USB RAW resource.

Parameters

- **serial_number** – The serial number of the instrument.
- **manufacturer_id** – The unique identification number of the manufacturer.

- **model_code** – The unique identification number of the product.
- **name** – Unique name given within Lantz to the instrument for logging purposes. Defaults to one generated based on the class name if not provided.
- **board** – USB Board to use
- **kwargs** – keyword arguments passed to the Resource constructor on initialize.

Return type *MessageBasedDriver*

write(*command*, *termination=None*, *encoding=None*)
Send command to the instrument.

Parameters

- **command** (*string.*) – command to be sent to the instrument.
- **termination** – termination character to override class defined default.
- **encoding** – encoding to transform string to bytes to override class defined default.

Returns number of bytes sent.

DEFAULTS = None
Default arguments passed to the Resource constructor on initialize. It should be specified in two layers, the first indicating the interface type and the second the corresponding arguments. The key COMMON is used to indicate keywords for all interfaces. For example:

```
{'ASRL':     {'read_termination': '\n',
              'baud_rate': 9600},
 'USB':      {'read_termination': \r'},
 'COMMON':   {'write_termination': '\n'}
}
```

Type dict[str, dict[str, str]]

MANUFACTURER_ID = None
The identification number of the manufacturer as hex code. :type: str | None

MODEL_CODE = None
The code number of the model as hex code. Can provide a tuple/list to indicate multiple models. :type: str | list | tuple | None

resource = None

Type pyvisa.resources.MessageBasedResource

resource_kwargs = None
keyword arguments passed to the resource during initialization. :type: dict

resource_name = None
The resource name :type: str

lantz.messagebased.**get_resource_manager**()
Return the PyVISA Resource Manager, creating an instance if necessary.

Return type visa.ResourceManager

lantz.foreign

Implements classes and methods to interface to foreign functions.

copyright 2015 by Lantz Authors, see AUTHORS for more details.

license BSD, see LICENSE for more details.

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

4.7.3 UI

lantz.ui.widgets

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

copyright 2015 by Lantz Authors, see AUTHORS for more details.

license BSD, see LICENSE for more details.

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

Parameters **`parent`** – parent widget.

class `lantz.ui.widgets.`**`FeatWidget`**
Widget to show a Feat.

class `lantz.ui.widgets.`**`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.widgets.`**`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.widgets.`**`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.widgets.`**`connect_setup`**(*parent*, *targets*, ***, *prefix=None*, *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.widgets.`**`initialize_and_report`**(*widget*, *drivers*, *register_finalizer=True*, *initializing_msg='Initializing ...'*, *initialized_msg='Initialized'*, *concurrent=True*, *dependencies=None*)

Initialize drivers while reporting the status in a QtWidget.

Parameters

- **`widget`** – Qt Widget where the status information is going to be shown.
- **`drivers`** – iterable of drivers to initialize.
- **`register_finalizer`** – register driver.finalize method to be called at python exit.
- **`initializing_msg`** – message to be displayed while initializing.
- **`initialized_msg`** – message to be displayed after successful initialization.
- **`concurrent`** – indicates that drivers with satisfied dependencies should be initialized concurrently.
- **`dependencies`** – indicates which drivers depend on others to be initialized. each key is a driver name, and the corresponding value is an iterable with its dependencies.

Returns the QThread doing the initialization.

`lantz.ui.widgets.`**`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.widgets.`**`request_new_units`**(*current_units*)

Ask for new units using a dialog box and return them.

Parameters **`current_units`** (*`Quantity`*) – current units or magnitude.

4.7.4 Support

lantz.stats

Implements an statistical accumulator

copyright 2015 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`**(*last, count, mean, std, min, max*)

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 2015 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
>>> getter(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.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.

4.8 Contributing

You are most welcome to contribute to Lantz with code, documentation and translations. Please read the following document for guidelines.

4.8.1 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.

4.8.2 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.
"""
```

4.8.3 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.
```

4.8.4 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*.

4.9 Contributing Drivers

The most straightforward way to contribute to Lantz is by submitting instrument drivers as you do not need to understand the whole structure of *Lantz*.

There are two ways:

- Clone the repo in github and do a pull request
- Upload your code to a gist,

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.

4.10 Contributing to the core

To contribute to the core, you need to clone the *Lantz* repository first.

4.10.1 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 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-3.4 install virtualenv
$ cd ~
$ virtualenv -p python3.4 --system-site-packages lantzenv
$ cd lantzenv
$ source bin/activate
```

and in Windows:

```
C:\Python34\Scripts\pip install virtualenv
cd  %USERPROFILE%\Desktop
C:\Python34\Scripts\virtualenv --system-site-packages lantzenv
cd lantzenv\Scripts
activate
```

and then install an editable package 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).

4.10.2 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.

4.10.3 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.

4.10.4 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.
"""
```

4.10.5 Submitting your changes

Changes must be submitted for merging as 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..

4.10.6 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.
```

4.10.7 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!
```

4.11 Agreeement

4.12 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.

4.13 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*