Transport Experiment Documentation

Release 1.5.0

Thomas C. Flanagan

June 14, 2014

CONTENTS

Contents:

CHAPTER

ONE

INTRODUCTION

Here is some random text. Blah, blah. And some more random text! Just can't get enough!

1.1 About the software

The Transport Experiment program was written (and is still being updated) for running transport measurements in the research group of Professor Nitin Samarth at the Pennsylvania State University.

The project was begun, and most of the code was written, by Thomas C. Flanagan in 2013.

1.2 Acknowledgements

I would like to thank the following people for enabling this project:

- David Hopper, for valuable contributions to the code;
- Anthony Richardella, for various suggests for the interface, some of which were sensible; and
- Abhinav Kandala, for giving me time to write it.

CHAPTER

TWO

GETTING STARTED

2.1 Running from source

2.1.1 The Python Interpreter

The primary dependency for the system is the Python interpreter. The code was written using **verson 2.7.5**. Note that any version greater or equal to than 3 will **not** work, mostly because the dependencies in the following sections have not all been updated to reflect changes in going from 2.x to 3.x. The most significant of these (and the most important in the context of this program) is PyVISA.

2.1.2 Dependencies

The following packages must be installed in order to run the program:

wxPython (2.8.12.1) wxPython is the graphical engine for the software. It was chosen for a number of reasons, some of which are listed below. #. It is fast, since it is merely a wrapper for the C-based wxWidgets library. #. Its API is relatively intuitive. #. It is popular, so it can be found in most (Debian-based) Linux repositories.

PyVISA (1.4) PyVISA is a library for interacting with GPIB and serial instruments.

numpy (1.7.1) and scipy (0.12.0) numpy and scipy are libraries for performing mathematical calculations. They are closely linked so downloads and documentation can usually be found together.

matplotlib (1.2.1) matplotlib is a plotting library which duplicates MATLAB in many ways.

The numbers in parentheses are, of course, versions. They are the versions under which the software was written and, therefore, the most likely to work as expected. This does not necessarily mean that other versions will not work. However, you *must* install versions which are compatible with Python 2.7.

2.1.3 Tools for managing documentation

As with any piece of software which multiple people may use and edit, it is essential to keep the documentation up to date and accurate. There are a number of tools which have been used to produce the documentation for this code in a clear and consistent way, and these are outlined in this section.

Python libraries

Pygments Pygments is a package for syntax highlighting which is used by Sphinx.

Sphinx This documentation starts out as a set of plain-text documents in the markup language reStructuredText, or reST. It is compiled into HTML, PDF, and HTML Help files using a package called Sphinx.

sphinx_wxoptimize sphinx_wxoptimize is a set of scripts for fixing the HTML Help packages produced by Sphinx so that it can be read in a reasonably decent manner by wxPython. You can find it at PyPI.

It should be noted that the first two of the above, Pygments and Sphinx, can be installed using `pip`, while the third, the script `sphinx_wxoptimize`, is already included in the `src/tools/docgen/manual` folder, so it should not need to be installed.

numpydoc The in-code documentation uses an extension to the standard Sphinx library called numpydoc, which allows for a cleaner-looking in-code syntax. Regenerating the API will crash Sphinx if this is not installed.

External applications

LaTeX The PDF manual is, of course, generated using `pdflatex`, so LaTeX must be installed.

Subversion Keeping the most recent version of the software available to everybody using it is a good idea, and for this project it is facilitated by Subversion. Download it. The most popular graphical client for Windows is TortoiseSVN. To use this software's automatic updater requires the command-line SVN tools. For Windows, the most common version is Slik Subversion, which can be downloaded here.

2.1.4 Tools

The most useful tool for this project is Eclipse, with the PyDev extension.

Instructions for installing this are coming soon.

2.2 Testing some code

Now let's test some junk.

```
def some_function(input_parameter):
    '''Print `input_parameter`, then attempt to convert it
    to a dictionary.
    '''
    print input_parameter
    try:
        return dict(input_parameter)
    except TypeError:
        print 'Cannot do that'
        return None
```

CHAPTER

THREE

PROJECT ORGANIZATION

The main software directory is called `Transport`, and this name is **very important**, since the code refers this name to turn relative paths into absolute paths. From here on out, all paths will be specified relative to this directory.

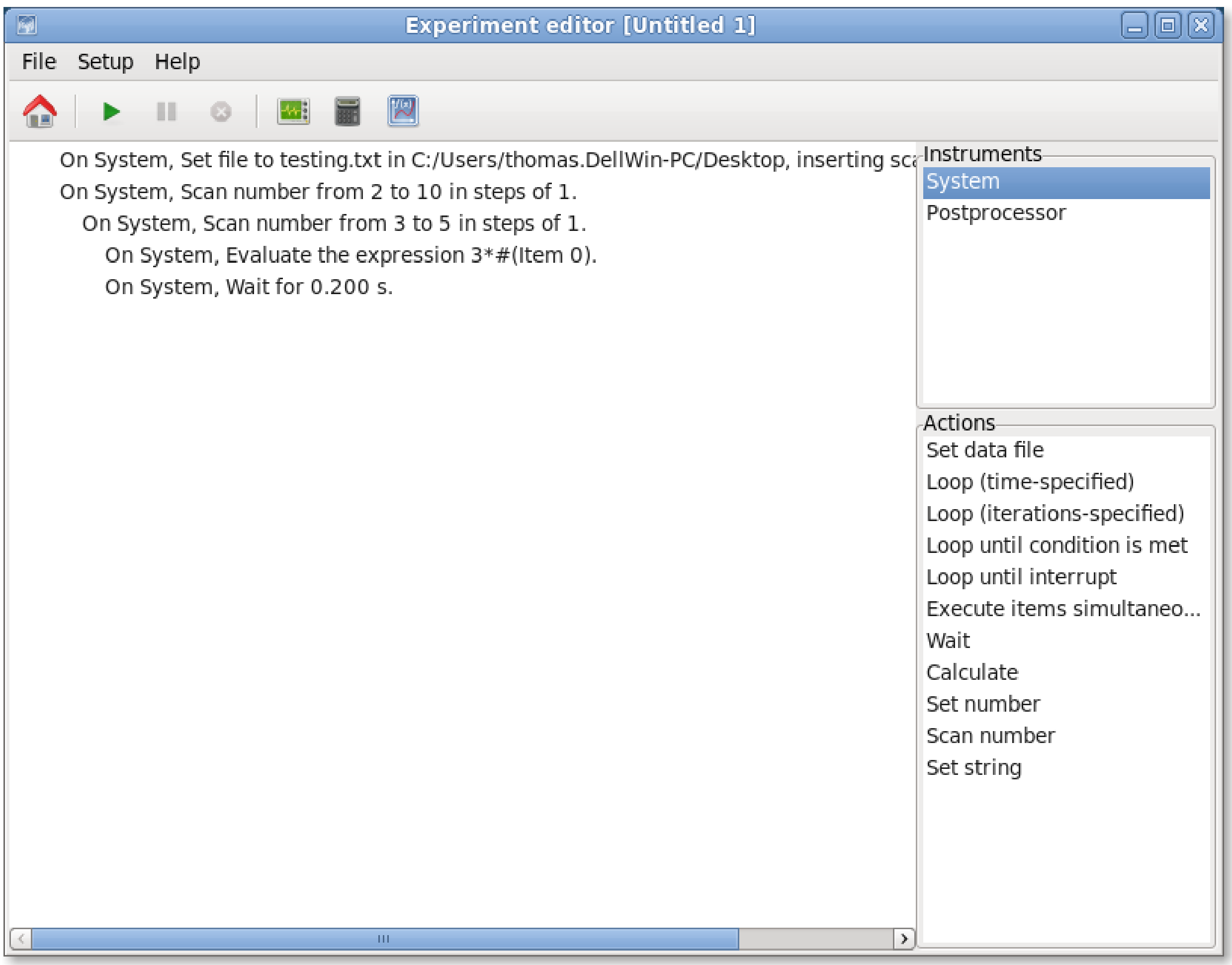

Figure 3.1: This is a random figure

CHAPTER

FOUR

EXTENDING THE CODE: INSTRUMENTS

Writing code for new instruments is usually quite straightforward (the main exception being cryostats, which take quite a bit of work). Below are the basic instructions for how to go about doing it and some pitfalls which would make the code fail.

4.1 Introduction

Since most instruments today follow a rather specific standard specified by IEEE, writing code to run them is quite simple. Nearly all instruments which can be connected to the computer via GPIB, USB, or RS232 implement VISA standards, and so the PyVISA module does all the work. Further, the commands follow a fairly standard form.

The first step in coding a new instrument is to create a module for it. For everything to work, there are a few requirements, which follow.

1. The module must be placed in the `src.instruments` package.
2. The module must contain a class which inherits from the `src.core.instrument.Instrument` class.
3. The class constructor (the `__init__()` method) must have the signature of an *Instrument*, it must call the its superclass constructor, and it must define all attributes of the class.
4. The new instrument must implement a class method `getRequiredParameters()` which should return a list of `src.core.instrument.InstrumentParameter` objects. Each of these objects indicates one attribute which must be specified in order for the instrument to work. For most GPIB instruments, the list will include only one element: the VISA resource address.
5. It must implement the methods `initialize()`, which opens a communication channel with the instrument, and py:meth:*finalize*, which closes said communication channel. Note that both of these methods are called automatically.
6. It must implement the `getActions()` method, which returns a list of `src.core.action.ActionSpec` objects indicating what actions the instrument can perform.

Let's consider these steps individually, with the SRS 830 Lock-In Amplifier as an example.

4.2 Module and class creation

The first is fairly obvious. The module name should be `src/instruments/srs830.py`.

The second step means that the class definition line should reference the `Instrument` object. The module header, of course, must import this object:

```
from src.core.instrument import Instrument
```

The class definition line should then look like this:

```
class SRS830(Instrument):
```

4.3 The instrument specification

Configuring an instrument for use will often require a bit of information about the instrument. These parameters are specified via instances of the `InstrumentParameter` class, which stores four attributes:

description A short, user-readable description of the parameter.

value The value of the parameter. The default is an empty string.

allowed The values which the parameter will accept, specified as a list of strings. If set to `None`, any value will be accepted.

formatString A string indicating how the value should be formatted. See *Format string syntax*.

For typical GPIB instruments, the only bit of such information will be its resource address, and so the `getRequiredParameters()` method will return a single-element list as follows:

```
@classmethod
def getRequiredParameters(cls):
    return [
        InstrumentParameter(
            description='Visa Address',
            value='',
            allowed=Instrument.getVisaAddresses,
            formatString='%s'
        )
    ]
```

This method simply returns the default for the `Instrument` subclass. The actual **value** for an **instance** is stored in the attribute `_spec`.

> **Warning:** Specifying a value for `allowed` makes no sense unless the value is a string.

4.4 Initialization and finalization

The constructor must be of the form:

```
def __init__(self, experiment, name='SRS830: Lock-in', spec=None):
    super(SRS830, self).__init__(experiment, name, spec)
    self._inst = None
    self._info = None
```

The requirement concerning the signature is, of course, implemented in the first line. Notice that all but `experiment` are optional (they have default values specified). The second line calls the parent class's constructor, and the third and fourth lines create the class's attributes, which will be assigned actual values when the instrument is initialized (at the beginning of the experiment's execution), as will be described next.

The fourth step requires that the instrument implement the `initialize()` and `finalize()` methods, which run at the beginning and the end of the experiment. Examples are the following:

```
def initialize (self):
    """Initialize the lock-in."""
    self._inst = visa.instrument(self._spec[0])
    info = ['Instrument: ' + self._name,
            'SRS 830: Lock-in amplifier',
            self._inst.ask('*IDN?')]
    self._info = '\n'.join(info)

def finalize (self):
    """Finalize the lock-in."""
    self._inst.close()
```

In the `initialize()` method, the `_inst` attribute is set to a `pyvisa.visa.Instrument` object. The argument to the constructor is the VISA resource address. The `_info` attribute is set to a three-line string describing the instrument, including its user-defined name, its model, and what it knows about itself.

In the `finalize()` method, the instrument communication channel is closed to free system resources.

4.5 Actions

Most instruments implement two types of actions: simple actions, which can set or read values, and scans, which repeat a simple action with multiple values. Regardless of its type, the action must define the following values:

experiment The `Experiment` object which owns the instrument. This will nearly always be the `Experiment` which owns the instrument, and so you can pass the attribute `self._expt`.

instrument The `Instrument` object which owns the action. This should nearly always be `self`.

description A short phrase describing the action in a way that users can understand.

inputs A list of parameters which will be sent to the instrument when it's time for it to perform the action.

outputs A list of parameters which the instrument will return once it has finished performing the action.

string A template string which will be filled in for turning the complete action sequence into strings for conveying information to the user.

method The (bound) method which will carry out the action. This will be discussed further later.

An action will be specified through a `collections.namedtuple` instance, `ActionSpec`, which has three attributes: `name`, a one-word name for the action, mainly for lookup purposes; `cls`, the `Action` class, or one of its subclasses, which will be used to construct the object; and `args`, a dictionary containing the keys listed above and their respective values.

An `ActionScan` object must have **one and only one** input, which should be a list of three-element tuples specifying the default range over which some quantity is varied. This range will be expanded, and the values will be passed sequentially to the method specified in the `ActionSpec`.

4.6 Parameters

Parameters are specified through a `collections.namedtuple` instance, `ParameterSpec`, which has attributes `name` and `args`. The first should be a short, single-word string to specify the parameter, and the second is a dictionary containing the following properties:

experiment The `src.core.experiment.Experiment` object which owns the action which owns the parameter.

description A short phrase describing the parameter in a way that users can understand.

value The default value for the parameter. It should be of the correct data type.

> **Note:** If the action is an `ActionScan`, the value should be specified as a list of tuples indicating the default scan components. For example, `'value': [(0.0, 1.0, 0.1), (1.0, 2.0, 0.5)]` would by default scan from 0.0 to 1.0 in steps of 0.1 and then from 1.0 to 2.0 in steps of 0.5.

binName The default name for the data storage bin to which the value will be saved, or `None` if it will not be saved by default.

binType The default type of data bin (either 'column' or 'parameter', or `None` if the data will not be saved by default).

formatString A string indicating how the value should be formatted. See *Format string syntax*.

> **Note:** If the action is an `ActionScan`, the formatString should end with '[]'

allowed A list containing the allowed values for the parameter. This only makes sense if the data type is a string.

4.7 The action syntax

The `getActions()` method should return a list of `ActionSpec` objects specifying all the actions which the instrument can perform (or, at least, all the actions which users of the instrument will *want* to perform).

The syntax for defining such an `ActionSpec` is as follows

```
ActionSpec(
    name='set_vref',
    cls=Action,
    args={
        'experiment': self._expt,
        'instrument': self,
        'description': 'Set reference voltage',
        'inputs': [
            ParameterSpec(
                name='vref',
                args={
                    'experiment': self._expt,
                    'description': 'Vref',
                    'formatString': '%.4f',
                    'binName': 'Vref',
                    'binType': 'parameter',
                    'value': 0,
                    'allowed': None,
                    'instantiate': False
                }
            )
        ],
        'string': 'Set the sine-out voltage to $vref.',
        'method': self.setReferenceVoltage
    }
)
```

The `name` values are very important. The name of the input parameter here is 'vref', and you can see that the same value occurs in the 'string' value for the `ActionSpec`. This is not a coincidence. When the software attempts to create informative strings about a given action, it will fill in the 'string' value, replacing all occurances of "${name}" with the value of the parameter specified by `{name}`.

> **Warning:** The values of `name` **must not contain spaces or special characters other than underscores**.

In the above code, the 'method' entry is set to `self.setReferenceVoltage()`. This is class method bound to the instance of the class whose `getActions()` method is called. Note the lack of parentheses at the end. This means that it is the **method itself**, and *not* the return value of the method, which is being put in that slot.

4.8 Defining the methods

Now, of course, to pass the `setReferenceVoltage()` to anything, the method must be defined in the class.

The first step in defining such a method is to find out the command which will induce the instrument to do what the `Action` wants. Referring to the manual for the SRS 830, we find that the command to set the reference voltage is "SLVL". Then the method to perform the action could be written like this:

```
def setReferenceVoltage (self, vref):
    self._inst.write('SLVL %.4f' % vref)
    return ()
```

The arguments to the method are `self`, which is required in all methods, and `vref`, which is the desired value for the reference voltage.

Warning: There are some important things to remember about the `vref` argument.

1. It has the same name as one of the `ParameterSpec` objects defined in the `getActions()` method above. It is the value of that parameter which will be substituted into this method, so if the names of the arguments to the method are not **exactly the same** as the names of the `ParameterSpec` objects defined in the 'inputs' bin of the relevant `ActionSpec`, the **software will crash**.
2. It is passed in whatever type is natural to the parameter. Since the reference voltage is a floating-point number, `vref` will be passed as a `float`. Therefore, since the SRS830 accepts ASCII string commands, the value must be turned into a string. That is why the string substitution occurs in the `write()` call above.
3. If the method is bound to the `ActionSpec` of an `ActionScan`, the method must have **one and only one** argument.

4.9 Format string syntax

Format strings follow format which is fairly standardized (it's actually the same as in LabVIEW). Such a string begins with the percent symbol. The final character depends on the data type:

type	character
integer	"d"
string	"s"
float	"f"
exponential	"e"

For the last two data types, both floating-point, the precision is specified by a period followed by the number of digits which should be printed after the decimal point. This bit should be between the percent sign and the data-type indicator. For example, to the format a number into a string of the form "2.012592e+02", use "%.6e".

There are actually considerably more options for customizing the string formatting, but they are less frequently used. More information can be found in a variety of places. For examples in the

Python language specifically, see the official documentation

4.10 Summary of potential problems

Here is a recap of the simple mistakes which would cause the program to crash.

1. The names of the `ParameterSpec` objects defined under 'inputs' for the relevant `ActionSpec` must **precisely match** the names of the arguments to the 'method' defined by said `ActionSpec`.
2. The names of `ParameterSpec` objects must also **precisely match** the values of the substitution strings (indicated by a dollar sign followed by the name) in the 'string' slot of the relevant `ActionSpec`.
3. The value of each argument to an instrument method needs to be converted to an appropriately formatted string if the natural type of the value is not already a string.
4. The value of 'allowed' for some `ParameterSpec` or `InstrumentParameter` should be `None` unless the value of the parameter should be a string and only certain values are allowed for that string, in which case 'allowed' should be a list of strings.
5. Values of the `name` attribute of instances of `ActionSpec` or `ParameterSpec` must **not** contain spaces or special characters (except the underscore).
6. For `ActionScan` objects, the 'formatString' of the `ParameterSpec` should end with "[]"
7. An `ActionScan` must have **one and only one** input, whose value is a list of three-element tuples which will be expanded into a range whose values will be passed sequentially into the method.

CHAPTER

FIVE

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*