Epics Applications using PyEpics

Release 1.0

Matthew Newville

April 25, 2012

CONTENTS

PyEpics Apps contains several Epics Applications written in python, using the pyepics module (see http://pyepics.github.com/pyepics/). Many of these are GUI Application for interacting with Epics devices through Channel Access. The programs here are meant to be useful as end-user applications, or at least as examples showing how one can build complex applications with PyEpics. Many of the applications here rely on wxPython, and some also rely on other 3rd party modules (such as Image and SQLAlchemy).

The list of applications should be expanding, but currently include:

CHAPTER

ONE

AREADETECTOR DISPLAY

An application to control and view images from an Epics Area Detector. The controls available in this viewer are minimal, but you can change mode, exposure time and frame rate, and start and stop the Area Detector. The image can be manipulated by zooming, rotating, and so on.

CHAPTER

TWO

STEP SCAN

A (work in progress!!) scanning program.

CHAPTER

THREE

STRIP CHART

A simple “live plot” of the recent history of a set of PV values, similar to the common Epics StripTool. Time ranges and ranges for Y values can be changed, and data can be saved to plain text files.

CHAPTER

FOUR

EPICS INSTRUMENTS

This application helps you organize PVs, by grouping them into instruments. Each instrument is a loose collection of PVs, but can have a set of named positions that you can tell it. That is, you can save the current set of PV values by giving it a simple name. After that, you can recall the saved values of each instrument, putting all or some of the PVs back to the saved values. The Epics Instrument application organizes instruments with tabbed windows, so that you can have a compact view of many instruments, saving and restoring positions as you wish.

CHAPTER

FIVE

SAMPLE STAGE

A GSECARS-specific GUI for moving a set of motors for a sample stage, grabbing microscope images from a webcam, and saving named positions.

CHAPTER

SIX

MOTOR SETUP

An application for setting up and saving the configuration of Epics Motors. For each opened motor, a full setup screen is shown in a tabbed notebook display. A paragraph for a Motors.template file can be saved for each motor, or copied to the system clipboard. In addition, you can save and read "known motor types" to a database, which holds most of the motor parameters.

The Local version uses a local sqlite database to store and read the known motor types, while the GSE version (which works only at GSECARS, of course) connects to a mysql server on the GSECARS network to store are read the known motor types.

CHAPTER

SEVEN

ION CHAMBER

This non-GUI application is synchrotron-beamline specific. It reads several settings for the photo-current of an ion chamber and calculates absorbed and transmitted flux in photons/sec, and writes these back to PVs (an associated .db file and medm .adl file are provided). The script runs in a loop, updating the flux values continuously.

CHAPTER

EIGHT

XRF COLLECTOR

This non-GUI application interacts with a small epics database to save data from a multi-element fluorescence detector. The script runs as a separate process, watching PVs and saving data from the detector on demand. Associated .db file and medm .adl file are provided.

8.1 Overview

An *Epics Instrument* is simply a grouping of low-level parameters (Process Variables) as exposed through Epics Channel Access. At first, this may not seem very interesting. However, Epics Instruments allows you to

The Epics Instruments application allows you to group these components into a logical group – an Instrument. Once an Instrument has been defined, you can then save and restore settings for this Instrument. Furthermore, these settings are automatcally saved in a single, portable file for later use.

Epics Channel Access gives a simple and robust interface to its lowest common unit – the Process Variable or PV. The Epics control system also provides sophisticated ways to express and manipulate complex devices, both physical and virtual. Creating such devices and defining their behavior is generally done by well-trained programmers. The application here uses a much simpler approach that can expose some categories of "Settings" that may need to changed en masse, and returned to at a later time.

As defined here, An Epics Instrument is simply a named collection of Epics Process Variables (PVs). The PVs do not need to be physically related to one another nor be associated with a single Epics Record or Device. Rather, an Instrument is defined at the level of the Epics Channel Access client, allowing a station scientist or engineer to use their own grouping of PVs as an abstract "Instrument". A simple example would be a pair of motors that work together to move some device.

In addition to a name and a set of PVs, an Instrument has a set of "named Positions". At any point, the current values of an Instruments PVs can be saved as its Position. And, of course, the named Positions can then be restored simply by selecting the Position.

8.2 Downloading and Installation

Many Epics Applications are available as GUI Application. For Windows, you can download and install from a binary installers below.

For Linux and Mac OS X, building and installing from source are currently the principle options (a Mac OS X App will be available soon).

8.2.1 Downloads

The latest source kits and installers will be at

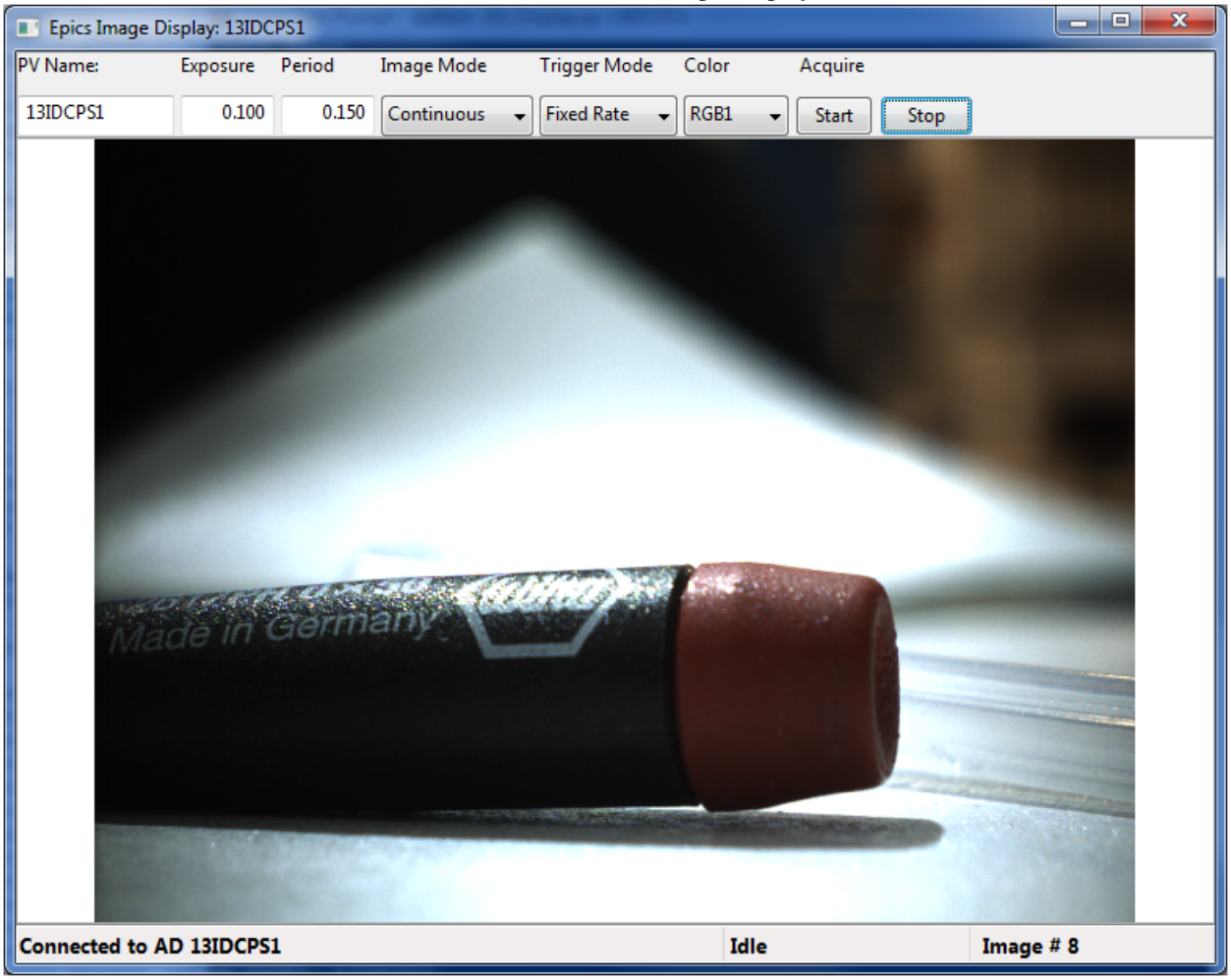
Download Option	Location
Windows Installers	cars_downloads
Source Kit	github_repo

8.3 Area Detector Display

Epics Area Detector Display is a wxPython GUI application for viewing images from an Epics Area Detector. This application requires wxPython, pyepics, numpy, and the Python Image Libary.

To run this application, simply run AD_Display.py at the command line:: python AD_Display.py

and enter the base name of the AreaDetector in the PV box. A sample display would look like this:



The fields and buttons on the top control several Area Detector settings, such as starting and stopping the acquisition.

8.4 StepScan: Epics Scans with Python

StepScan is a library and (soon...) GUI application for running simple to moderately complex step scans of Epics motors, detectors, and general variables. Many concepts are borrowed from the Epics SScan record and from Spec, but neither of those are used by StepScan – the scans happen in the python client application.

With Epics, nearly any Process Variable can be scanned or counted during a scan, and so a Step Scan can be a fairly complex thing. Fortunately, simple things are still easy to do. A description of some terms in the *Step Scan Concepts* section may be helpful.

8.4.1 StepScan Installation

Dependencies, Installation

This application needs pyepics, numpy, and wxPython. In order to save data to HDF5, the HDF5 library, and h5py module (available at http://code.google.com/p/h5py/) will be required.

To install, use:

```
python setup.py install
```

from the StepScan folder. A windows installer will be made available soon

Running the StepScan GUI

To run the Epics StepScan GUI, use:

```
python pyepics_stepscan.py
```

or click on the icon.

Using StepScan from Python

To use StepScan from Python scripts,

8.4.2 Step Scan Concepts

A step scan is simply a **for loop** that iterates a collection of **Positioners** (any Epics PV – not restricted to Motors) through a set of pre-determined position values. At each position, a set of detectors are **triggered** (which is to say, collection is started), and then waited upon until they announce they are complete. At that point a collection of **Counters** (again, any Epics PV, though for those holding waveform values see the notes below) are read, and the loop starts over, moving on to the next point in the list of positions.

A StepScan also contains several other components::

- extra_pvs: a set of PVs to record once (or a few times) for each scan, but not at every point in the scan.
- breakpoints: a series of points in the scan at which data collected so far is dumped to file, and perhaps other tasks are done.
- pre_scan: a set of functions to call just prior to beginning the scan.
- post_scan: a set of functions to call just after the scan completes.
- at_break: a set of functions to call at each breakpoint (after the data is written to disk).

Positioners

Positioners are what is changed in the scan – the dependent variables. The can be represented by any Epics PV, such as Motors, temperatures, currents, count times, or a combination of these. Scans can have multiple positioners, either moving in tandem to make a complex motion in the positioner space, or independently to make a mesh in the positioner space.

In addition to a PV, each positioner will have an array of positions which it should take during the scan. The Positioner holds the full array positions. There are methods available for creating this from Start, Stop, Step, and Npts parameters for simple linear scans, but you can also give it a non-uniform array,

If you have multiple positioners, they must all have an array of positions that is the same length.

Triggers

A Trigger is something that starts a detector counting. The general assumption is that these start integrating detectors with a finite count time (which may have been set prior to the scan, or set as one of the Positioners). The scan waits for these triggers to announce that they have completed (ie, the Epics "put" has completed). That means that not just any PV can be used as a Trigger – you really need a trigger that will finish.

For many detectors using standard Epics Records (Scalers, MCAs, multi-element MCAs, AreaDetectors), using a Detector (see below) will include the appropriate Trigger.

Counters

A Counter is any PV that you wish to measure at each point in the scan.

For many detectors using standard Epics Records (Scalers, MCAs, multi-element MCAs, AreaDetectors), using a Detector (see below) will automatically include a wide range of Counters.

Counters for waveforms

Currently, these are not directly supported, mostly as it is not yet determined how to save this data into a plain ASCII file. If more complex file formats are used, these could be supported.

Note that a Trigger for an AreaDetector can be included, which might cause it to save data through its own means.

Detectors

Detectors are essentially a combination of Triggers and Counters that represent a detector as defined by one of the common Epics detector records. These include Scalers, MCAs, multi-element MCAs, and AreaDetectors.

Extra PVs

Extra PVs are simply Epics PVs that should be recorded once (or occasionally) during a scan, but but not at every point in the scan. These might include configuration information, detector and motor settings, and ancillary data like temperatures, ring current, etc.

These should be supplied as a list of *(Label, PVname)* tuples. The values for these will be recorded at the beginning of each scan, and at each breakpoint.

Breakpoints

Breakpoints are points in the scan at which the scan pauses briefly to to read values for Extra PVs, write out these and the data collected so far to disk, and run any other user-defined functions.

A typical short, one dimensional scan will not need any breakpoints – the data will be written at the end of the scan. On the other hand, a two dimensional mesh scan may want to set a breakpoint at the end of each row, so that the data collected to date can be read from the datafile.

To set breakpoints, just put the indices of the points to break after (starting the counting from 0) into a *scan.breakpoints* list.

Users can add their own functions to be run at each breakpoint as well.

Pre- and Post-Scan functions

These functions are run prior to and just after the scan has run. Detectors and Positioners may add their own `pre_scan()` and `post_scan()` methods, for example to place a detector in the right mode for scanning.

8.4.3 Using StepScan

This section describes how to use a StepScan from python scripts. See *Step Scan Concepts* for a description of terms and basic concepts.

Defining a New Scan

To define a new scan, you create a scan object, set up the Positioners, Detectors, and Counters for the scan, add optional extra features such a as extra_pvs and breakpoints, and then run the scan.

Running a Scan

8.4.4 Spec-like Scans with StepScan

Spec is a commonly used program for collecting data at synchrotrons, as it can not only communicate with Motors and Detectors, but can also work in the geometries needed to run diffractometers, and includes a convenient macro language. Spec is not without limitations, and a reasonable goal of StepScan would be to replace some uses of Spec. Of course, Python can replace the macro language, and Epics Channel Access can control Motor and Detectors. To better enable a transition of some scanning away from Spec and toward Python, StepScan has a `spec_emulator` module that provides Spec-like functionality, with scanning methods `ascan()`, `dscan()`, `a2scan()`, `d2scan()`, `lup()`, and `mesh()` that closely match the Spec routines of the same name.

Setting up and Configuring the spec emulation layer

To emulate Spec, one creates a `SpecScan` object:

```
>>> from epicsapps.stepscan import SpecScan
>>> spec = SpecScan()
```

This creates an empty SpecScan object, which does not have any configuration information about Motors and Detectors to use during a scan. Motors can be added to the configuration with:

```
>>> spec.add_motors(x='XXX:m1', y='XXX:m2', theta='XXX:m3')
```

where you can add as many *label=PV_NAME* keyword arguments as you like. Later on, when scanning, you'll use the labels to mean the underlying PVs. As with all of StepScan, the PVs are not required to be motors.

We'll also need to add some detectors. StepScan supports many kinds of detectors, but for simple scans with point-detectors, an Epics Scaler record usually suffices, so we'll start with that:

```
>>> spec.add_detector('XXX:scaler1', kind='scaler', nchan=8, use_calc=False)
```

By saying *kind='scaler'*, StepScan will know what to use for a Trigger, how to set the dwell time, and what Counters to count – all the channels 1 through 8 that have a non-blank name will be collected. There is also the option to set *use_calc=True* to use the calc record associated with the Scan Record, which can be used to support offset values and simple calculations.

As with other parts of StepScan, we can add *extra PVs* – values to be recorded at the start of each scan, with a list of (description, pvname) tuples:

```
>>> spec.add_extra_pvs((('Ring Current', 'S:SRcurrentAI.VAL'),
                        ('Ring Lifetime', 'S:SRlifeTimeHrsCC.VAL')))
```

We haven't yet set the name of the output file. This is still a work-in-progress, but currently, the Spec file is not emulated in detail, but an ASCII file is written to for each scan. We can change this anytime with:

```
>>> spec.filename = 'myoutput.dat'
```

At this point, we have a fully configured (if minimal) Spec emulator, and can start running scans. It is likely

Using the spec emulation layer

To perform a simple absolute scan, we can use `ascan()`, which would be as simple as:

```
>>> spec.ascan('x', 0, 0.5, 51, 0.5)
```

which scans Motor 'x' between 0 and 0.5 in 21 steps, counting for 0.5 second per point.

Other supported scan routines are `dscan()`, which is like `ascan()` but with start and stop positions relative to the current position:

```
>>> spec.dscan('x', -0.1, 0.1, 41, 0.25)
```

The routines `a2scan()` and `d2scan()` simultaneously moves 2 Motors (in absolute and relative coordinates, respectively):

```
>>> spec.a2scan('x', 0, 1.0, 'y', 0, 0.2, 21, 0.5)
>>> spec.d2scan('x', -0.5, 0.5, 'y', -0.1, 0.1, 21, 0.5)
```

That is, these move along a line in 'x'-'y' space.

The routines `a3scan()` and `d3scan()` simultaneously moves 3 Motors (in absolute and relative coordinates, respectively):

```
>>> spec.a3scan('x', 0, 1.0, 'y', 0, 0.2, 'theta', 10.0, 10.1, 21, 0.5)
```

Finally, `mesh()` will make a 2-dimensional mesh, or map:

```
>>> spec.mesh('x', 10, 11, 21, theta', 10.0, 10.1, 11,  0.25)
```

This will make a 21 x 11 pixel map, moving 'x' between 10 and 11 for each 'theta' value, so that 'theta' scans slowly.

8.4.5 Using the StepScan GUI

8.5 Strip Chart Display

StripChart is a wxPython GUI application for viewing time traces of PVs as a strip chart. It feature interactive graphics, with click-and-drag zooming, updating the plotted time range, saving figures as high-quality PNGs, and saving data to ASCII files. Stripchart is inspired somewhat by the classic Epics Stripchart application written with X/Motif, but has many differences.

8.5.1 Dependencies, Installation

This application needs pyepics, numpy, matplotlib, and wxPython.

It also the wxmplot plotting library, which can be found at http://pypi.python.org/pypi/wxmplot/, with development versions at http://github.com/newville/wxmplot/ and may be installed wth:

```
easy_install -U wxmplot
```

Installation of the striphart can be done with:

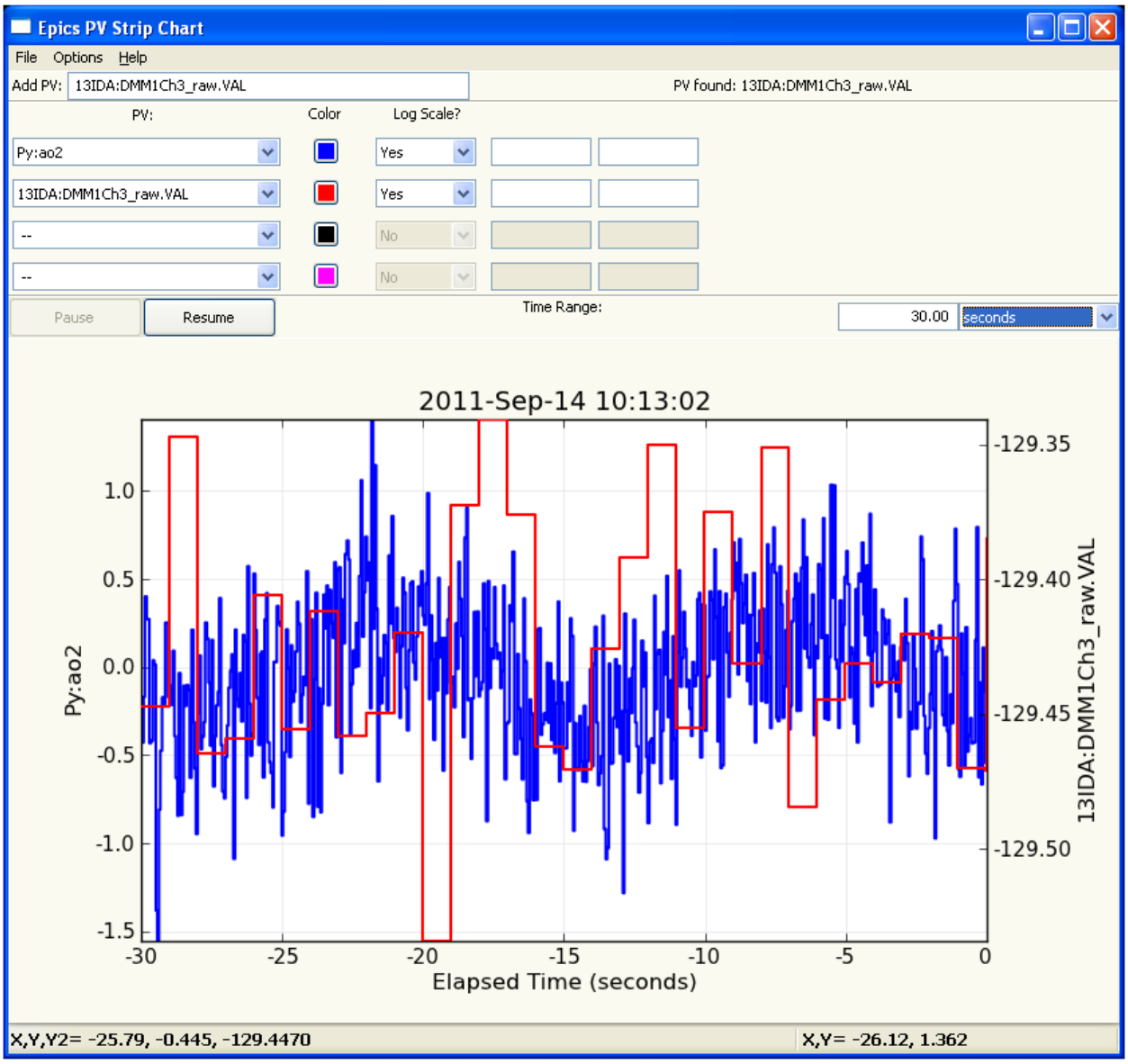
```
python setup.py install
```

8.5.2 Running Stripchart

To run this application, simply run stripchart.py at the command line:

```
python pyepics_stripchart.py
```

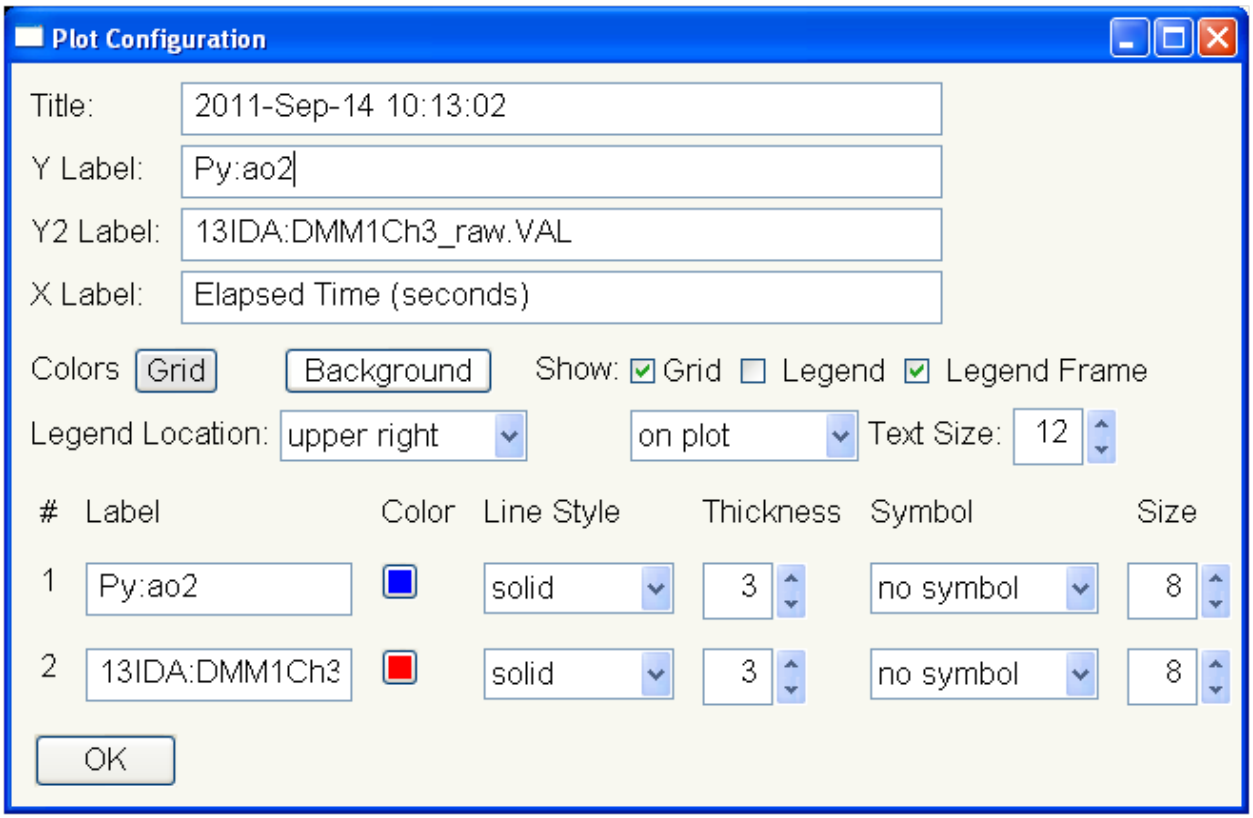
and enter the base name of the PVs to follow. A sample display would look like this:



8.5.3 Usage

Plot details can be adjusted from the configuration form, available from the Options Menu, and shown below. From this frame, you can adjust trace colors, symbols, line width and style, symbol size and styles, axes labels, and the contents and location of a plot legend. Text for titles, axes labels, and legend can include latex strings for math/Greek characters.

From the main plot, Ctrl-C works to copy to the system clipboard, and Ctrl-P will open a print dialog.



8.6 Using Epics Instruments

Epics Instruments is a GUI application (using wxPython) that lets any user:

- Organize PVs into Instruments: a named collection of PVs
- Manage Instruments with modern Notebook-tab interface.
- Save Positions for any Instrument by name.
- Restore Positions for any Instrument by name.
- Remember Settings for all definitions into a single file that can be loaded later.
- Multiple Users can be using multiple instrument files at any one time.

It was originally written to replace and organize the multitude of similar MEDM screens that appear at many workstations using Epics.

8.6.1 Dependencies, Installation

This application needs pyepics, numpy, wxPython, and SQLAlchemy (available at http://www.sqlalchemy.org/).

To install, use:

```
python setup.py install
```

from the Instruments folder. A windows installer may be available.

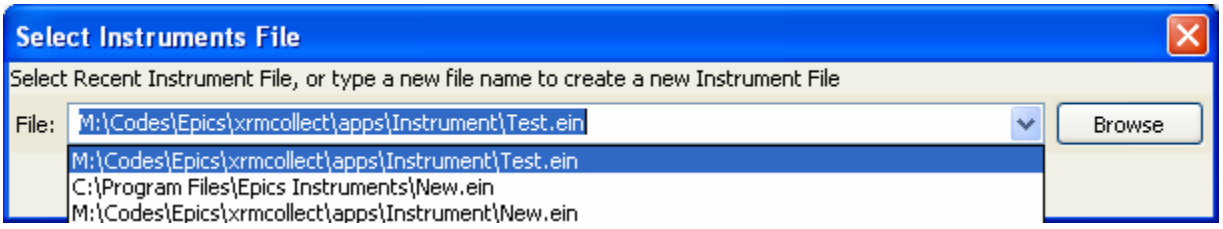
8.6.2 Getting Started

To run Epics Instruments, use:

```
python pyepics_instruments.py
```

or click on the icon.

A small window to select an Epics Instrument File, like this



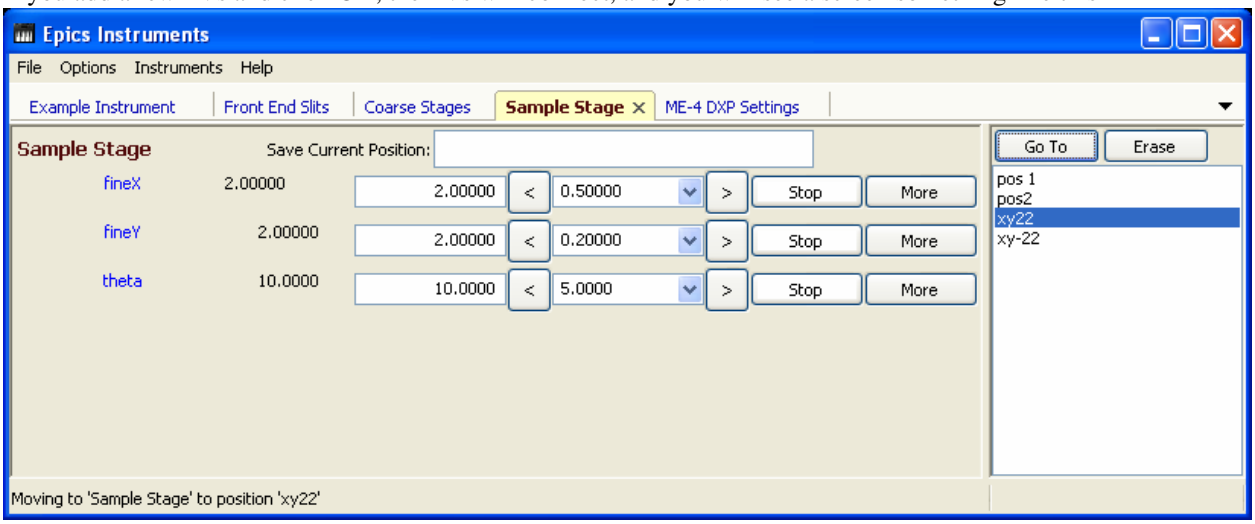
If this is your first time using the application, choose a name, and hit return to start a new Instrument File. The next time you run Epics Instruments, it should remember which files you've recently used, and present you with a drop-down list of Instrument Files. Since all the definitions, positions, and settings are saved in a single file, restoring this file will recall the earlier session of instrument definitions and saved positions.

An Epics **Instrument** is a collection of PVs. Each Instrument will also have a collection of **Positions**, which are just the locations of all the PVs in the instrument at the time the Position was saved. Like a PV, each Instrument and each Position for an Instrument has a unique name.

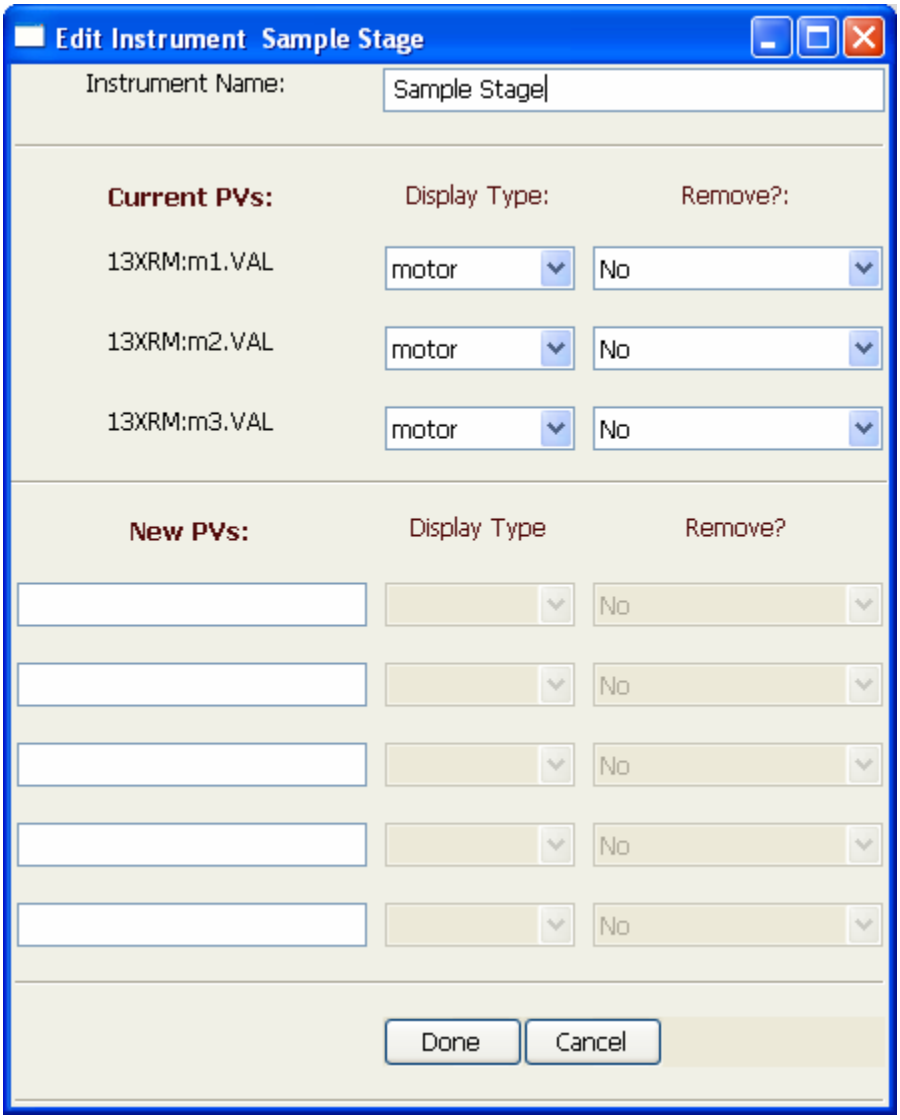
8.6.3 Defining a New Instrument

To define a new Instrument, select **Create New Instrument** from the Instruments Menu. A screen will appear in which you can name the instrument and the PVs that belong to the Instrument.

If you add a few PVs and click OK, the PVs will connect, and you will see a screen something like this



8.6.4 Editing an Exisiting Instrument



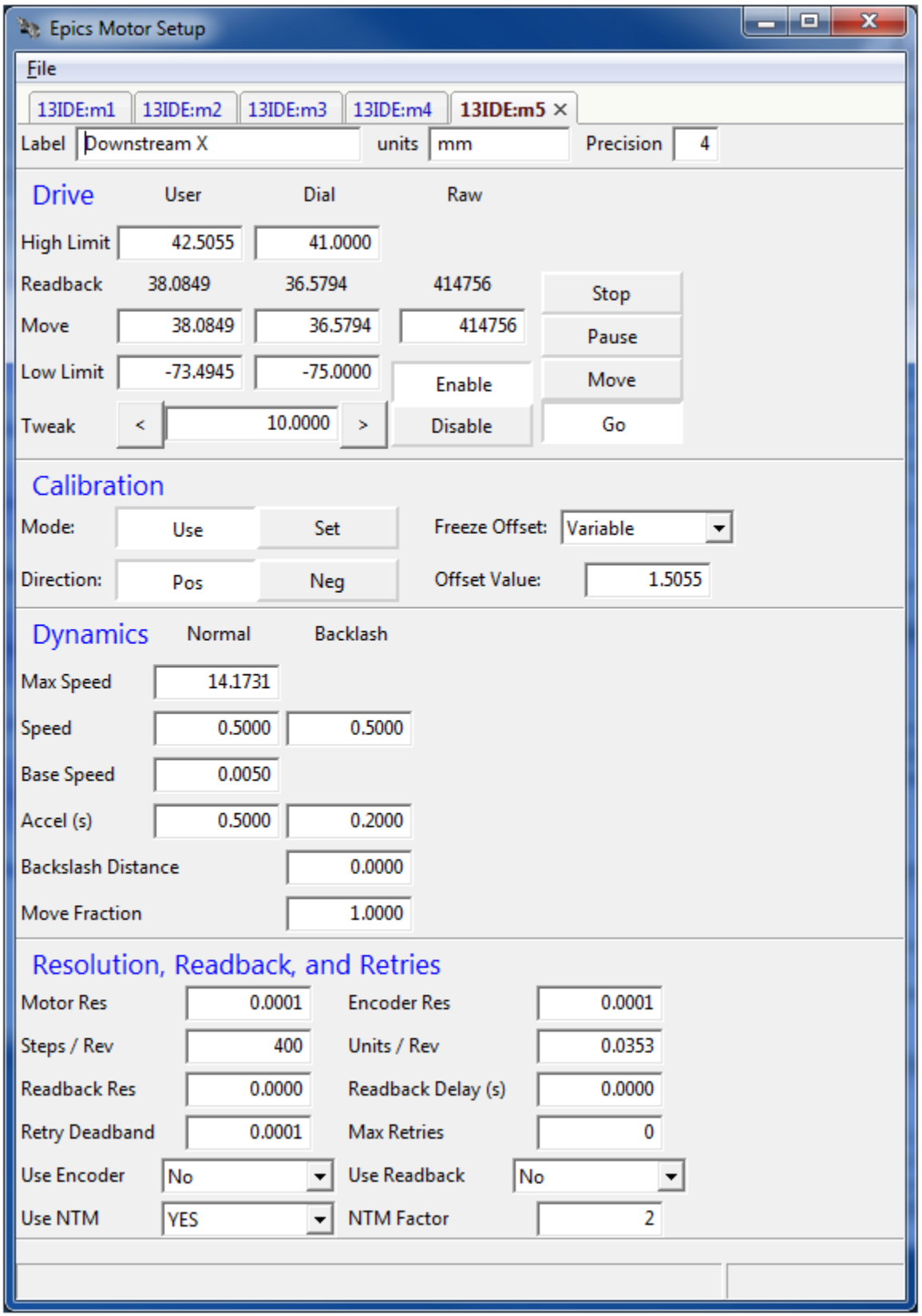
8.6.5 The Instrument File

All the information for definitions of your Instruments and their Positions are saved in a single file – the Instruments file, with a default extension of '.ein' (Epics INstruments). You can use many different Instrument Files for different domains of use.

The Instrument File is an SQLite database file, and can be browsed and manipulated with external tools. Of course, this can be a very efficient way of corrupting the data, so do this with caution. A further note of caution is to avoid having a single Instrument file open by multiple applications – this can also cause corruption. The Instrument files can be moved around and copied without problems.

8.7 Using Epics Motor Setup

Epics Motor Setup is a fairly simple GUI application (using wxPython) for setting up Epics Motors. A full configuration screen is shown for each motor, using a Notebook display:



The real advantage of this Application is in two simple features:

1. being able to write a **motors.template** file for all the open motors.

2. being able to read and save motor settings for future use to a database. By default, a local SQLite database is used, but users at GSECARS can also use a GSE-wide database.

To save the **motors.template** information, simply type Ctrl-T to copy the full template paragraph to the system clipboard, then copy into the appropriate file. A simple example is:

```
file "$(CARS)/CARSApp/Db/motor.db"
{
pattern
{P,  M,  DTYP,  C,  S,  DESC,  EGU,  DIR,  VELO, VBAS, ACCL, BDST,
BVEL, BACC, S    REV, UREV, PREC, DHLM, DLLM}
# VAL=1.999519, OFF=0.799999, NTM=1
{13IDE:, m1, "OMS VME58", 0, -1, "Upstream Y", mm, Pos, 0.200000, 0.005000, 1.00    0000, 0.000000, 0
# VAL=2.001424, OFF=-1.000026, NTM=1
{13IDE:, m2, "OMS VME58", 0, -1, "Inboard Y", mm, Pos, 0.200000, 0.005000, 1.000    000, 0.000000, 0.
# VAL=1.999519, OFF=1.069979, NTM=1
{13IDE:, m3, "OMS VME58", 0, -1, "Outboard Y", mm, Pos, 0.200000, 0.005000, 1.000000, 0.000000, 0.200
# VAL=11.035400, OFF=1.505489, NTM=1
{13IDE:, m4, "OMS VME58", 0, -1, "Upstream X", mm, Pos, 0.500000, 0.005000, 0.500000, 0.000000, 0.500
# VAL=38.084885, OFF=1.505479, NTM=1
{13IDE:, m5, "OMS VME58", 0, -1, "Downstream X", mm, Pos, 0.500000, 0.005000, 0.500000, 0.000000, 0.5
}
```

A few things to note here are:

1. You may need to change "$(CARS)/CARSApp/Db/motor.db" to point to the correct location of the motor.db file.
2. Though the motor "device type" and "card" are filled out, the "slot" is not available as a PV, and so is not filled out.
3. The VAL (User Value), OFF (Dial Offset) and NTM field are saved for each motor as a comment.

8.8 XRF Collector

XRF Collector provides a simple Epics interface control of an x-ray fluorescence detector. The main point of showing it here is as an example of using a python process to interact with a custom Epics database to provide a customized way to acquire data.

8.9 Ion Chamber

This application, like the XRF Collector, provides a simple Epics interface control of a non-trivial device. Again, the main point of showing it here is as an example of using a python process to interact with a custom Epics database to provide a customized way to acquire data. While the standard Epics solution would be to write a state-notation-language program that runs in an IOC, the approach here minimizes Epics coding to writing a simple database, and putting all the logic and control into a python script that runs as a long-running process along-side an IOC process.

PYTHON MODULE INDEX

S

INDEX

S