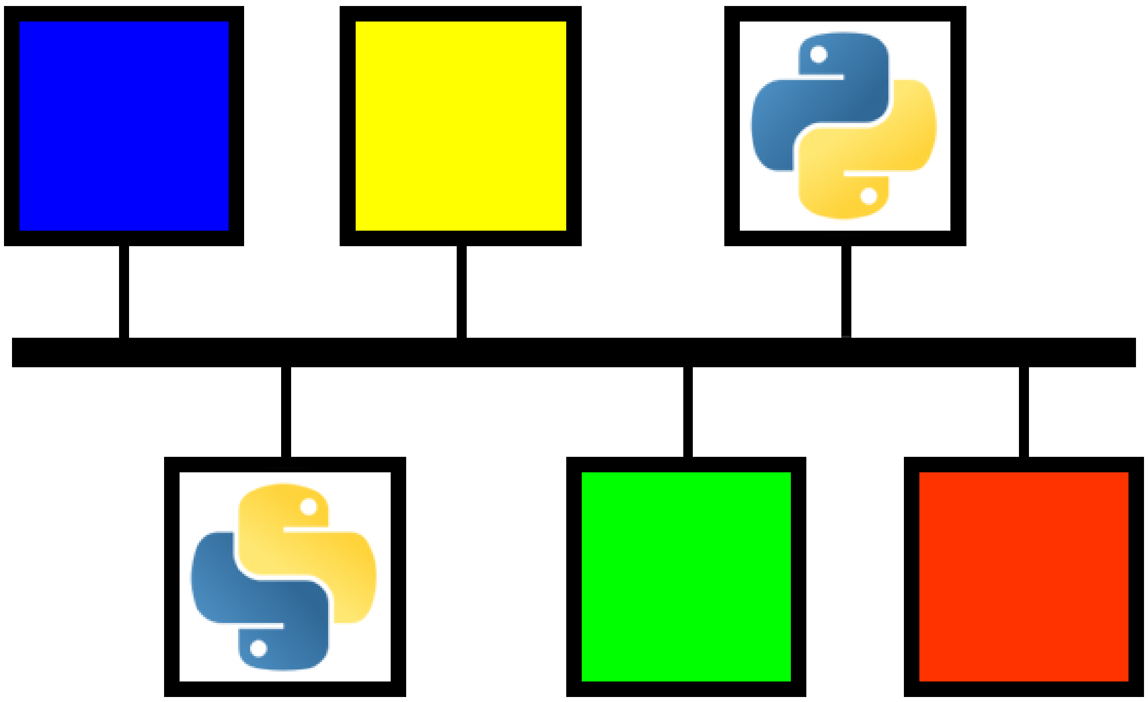


PyEpics: Python Epics Channel Access

Release 3.2.5

Matthew Newville

May 21, 2016

PyEpics is an interface for the Channel Access (CA) library of the Epics Control System to the Python Programming language. The pyepics package provides a base `epics` module to python, with methods for reading from and writing to Epics Process Variables (PVs) via the CA protocol. The package includes a fairly complete, thin layer over the low-level Channel Access library in the `ca` module, and higher-level abstractions built on top of this basic functionality.

The package includes a simple, functional approach to CA similar to EZCA and the Unix command-line tools with functions in the main `epics` package including `epics.caget()`, `epics.caput()`, `epics.cainfo()`, and `epics.camonitor()`. There is also a `pv.PV` object which represents an Epics Process Variable as an easy-to-use Python object. Additional modules provide even higher-level programming support to Epics. These include groups of related PVs in `device.Device`, a simple method to create alarms in `alarm.Alarm`, and support for saving PVs values in the `autosave` module. Finally, there is support for conveniently tying epics PVs to wxPython widgets in the `wx` module.

In addition to the Pyepics library described here, several applications built with pyepics are available at http://github.com/pyepics/epicsapps/. See http://pyepics.github.com/epicsapps/ for further details.

CHAPTER

ONE

DOWNLOADING AND INSTALLATION

1.1 Prerequisites

This package requires Python version 2.7, 3.4, or 3.5. It should work with Python 2.6, 3.2, or 3.3, but these are no longer being tested regularly.

In addition, version 3.14 of the EPICS Channel Access library (v 3.14.8 or higher, I believe) is required. More specifically, the shared libraries libCom.so and libca.so (or Com.dll and ca.dll on Windows) from *Epics Base* are required to use this module. Using version 3.14.12 or higher is recommended – some of the features for 'subarray records' will only work with this 3.14.12 and higher. For Linux and Mac OS X (and other Unix-like systems), these dynamic libraries are assumed to be available, and findable by Python at runtime. This may mean you have to set environmental variables such as PATH, LD_LIBRARY_PATH, or DYLD_LIBRARY_PATH or using Linux tools such as ldconfig.

For 32-bit Python on 32-bit or 64-bit Windows, pre-built DLLs from 3.14.12 (patched as of March, 2011) are included and installed so that no other Epics installation is required to use the modules.

For 64-bit Python on 64-bit Windows, pre-built DLLs from 3.14.12.4 (patched as of January, 2015) are also included. Support for 64-bit Python on Windows was new in version 3.2.4, and requires Python version 2.7.9.

The Python numpy module is highly recommended, though it is not strictly required. If available, it will be used to convert EPICS waveforms values into numpy arrays.

1.2 Downloads and Installation

The latest stable version of the PyEpics Package is 3.2.5. Source code kits and Windows installers can be found at either pyepics PyPI or pyepics CARS downloads site. With Python Setup Tools now standard for Python 2.7 and abouve, the simplest way to install the pyepics is with:

```
pip install pyepics
```

If you're using Anaconda, there are a few conda channels for pyepics, including:

```
conda install -c https://conda.anaconda.org/newville pyepics
```

1.3 Getting Started, Setting up the Epics Environment

In order for PyEpics to work at correctly, it must be able to find and load the Channel Access dynamic library (*libca.so*, *libca.dylib*, or *ca.dll* depending on the system). This dynamic library needs to found at runtime.

There are a few ways to specify how to find this library:

1. set the environmental variable `PYEPICS_LIBCA` to the full path of the dynamic library, for example:

```
> export PYEPICS_LIBCA=/usr/local/epics/base-3.14.12.1/lib/linux-x86/libca.so
```

2. set the environmental variables `EPICS_BASE` and `EPICS_HOST_ARCH` to point to where the library was built. For example:

```
> export EPICS_BASE=/usr/local/epics/base-3.14.12.1
> export EPICS_HOST_ARCH=linux-x86
```

   will find the library at /usr/local/epics/base-3.14.12.1/lib/linux-x86/libca.so.

3. Put the dynamic library somewhere in the Python path. A convenient place might be the same `site-packages/pyepics library` folder as the python package is installed.

To find out which CA library will be used by pyepics, use:

```
>>> import epics
>>> epics.ca.find_libca()
```

which will print out the full path of the CA dynamic library that will be used.

Note for Windows users: The needed CA DLLs (ca.dll and Com.dll) are included in the installation kit, and should be automatically installed to where they can be found at runtime (following rule 3 above). If they are not found (or another version is found that conflicts), you should copy them to a place where they can be found, such as the Python DLLs folder, which might be something like `C:\Python27\DLLs`.

For more details, especially about how to set paths for LD_LIBRARY_PATH or DYLD_LIBRARY_PATH on Unix-like systems, see the INSTALL file.

With the Epics library loaded, it will need to be able to connect to Epics Process Variables. Generally, these variables are provided by Epics I/O controllers (IOCs) that are processes running on some device on the network. If you're connecting to PVs provided by IOCs on your local subnet, you should have no trouble. If trying to reach further network, you may need to set the environmental variable `EPICS_CA_ADDR_LIST` to specify which networks to search for PVs.

1.4 Testing

Automated, continuous unit-testing is done with the TravisCI (https://travis-ci.org/pyepics/pyepics) for Python 2.7, 3.4, and 3.5 using an Epics IOC running in a Docker image. Many tests located in the *tests* folder can also be run using the script `tests/simulator.py` as long as the Epics database in `tests/pydebug.db` is loaded in a local IOC. The following systems were tested for 3.2.5 were tested locally, all with Epics base 3.14.12.1 or higher:

Host OS	Epics HOST ARCH	Python	Test Status
Linux	linux-x86	2.6 32bit	pass
Linux	linux-x86	2.7.3 32bit	pass
Linux	linux-x86_64	2.7.8 64bit	pass
Linux	linux-x86_64	3.4.1 64bit	pass
Mac OSX	darwin-x86	2.7.8 64bit	pass
Windows	win32-x86	2.6.6 32bit	pass
Windows	win32-x86	2.7.8 32bit	pass
Windows	win32-x86	3.4.1 32bit	pass
Windows	windows-x64	2.7.9 64bit	pass
Windows	windows-x64	3.4.2 64bit	pass

Testing Notes:

1. tests involving subarrays are known to fail with Epics base earlier than 3.14.11.

2. The autosave module relies on the 3rd part extension pyparsing, which seems to not work correctly for Python3, and has not been included in these tests.
3. The wx module is not automatically tested.

1.5 Development Version

The PyEpics module is still under active development, with enhancements and bug-fixes are being added frequently. All development is done through the pyepics github repository. To get a read-only copy of the latest version, use one of:

```
git clone http://github.com/pyepics/pyepics.git
git clone git@github.com/pyepics/pyepics.git
```

1.6 Getting Help

For questions, bug reports, feature request, please consider using the following methods:

1. Send email to the Epics Tech Talk mailing list. You can also send mail directly to Matt Newville <newville@cars.uchicago.ed>, but the mailing list has many Epics experts reading it, so someone else interested or knowledgeable about the topic might provide an answer. Since the mailing list is archived and the main mailing list for Epics work, a question to the mailing list has a better chance of helping someone else.
2. Create an Issue on http://github.com/pyepics/pyepics. Though the github Issues seem to be intended for bug tracking, they are a fine way to catalog various kinds of questions and feature requests.
3. If you're sure you've found a bug in existing code, or have some code you think would be useful to add to PyEpics, and you're familiar with git, make a Pull Request on http://github.com/pyepics/pyepics.

1.7 License

The PyEpics source code, this documentation, and all material associated with it are distributed under the Epics Open License:

The epics python module was orignally written by

> Matthew Newville <newville@cars.uchicago.edu> CARS, University of Chicago

There have been several contributions from many others, notably Angus Gratton <angus.gratton@anu.edu.au>. See the Acknowledgements section of the documentation for a list of more contributors.

Except where explicitly noted, all files in this distribution are licensed under the Epics Open License.:

Copyright 2010 Matthew Newville, The University of Chicago. All rights reserved.

The epics python module is distributed subject to the following license conditions: SOFTWARE LICENSE AGREEMENT Software: epics python module

> 1. The "Software", below, refers to the epics python module (in either source code, or binary form and accompanying documentation). Each licensee is addressed as "you" or "Licensee."
>
> 2. The copyright holders shown above and their third-party licensors hereby grant Licensee a royalty-free nonexclusive license, subject to the limitations stated herein and U.S. Government license rights.

In plain English, this says that there is no warranty or gaurantee that the code will actually work, but you can do anything you like with this code except a) claim that you wrote it or b) claim that the people who did write it endorse your use of the code. Unless you're the US government, in which case you can probably do whatever you want.

1.8 Acknowledgments

PyEpics was originally written and is maintained by Matt Newville <newville@cars.uchicago.ed>. Many important contributions to the library have come from Angus Gratton while at the Australian National University. Several other people have provided valuable additions, suggestions, or bug reports, which has greatly improved the quality of the library: Ken Lauer, Robbie Clarken, Daniel Allen, Michael Abbott, Daron Chabot, Thomas Caswell, Georg Brandl, Niklas Claesson, Jon Brinkmann, Marco Cammarata, Craig Haskins, Pete Jemian, Andrew Johnson, Janko Kolar, Irina Kosheleva, Tim Mooney, Eric Norum, Mark Rivers, Friedrich Schotte, Mark Vigder, Steve Wasserman, and Glen Wright.

CHAPTER

TWO

PYEPICS OVERVIEW

The python `epics` package consists of several function, modules, and classes to interact with EPICS Channel Access. The simplest approach uses the functions `caget()`, `caput()`, `cainfo()`, `camonitor()`, and `camonitor_clear()` within the top-level *epics* module. These functions are similar to the standard command line utilities and to the EZCA library interface, and are described in more detail below.

To use the `epics` package, import it with:

```
import epics
```

The main components of this module include

- functions `caget()`, `caput()`, `camonitor()`, `camonitor_clear()`, and `cainfo()` as described below.
- a `ca` module, providing the low-level Epics Channel Access library as a set of functions.
- a `PV` object, giving a higher-level interface to Epics Channel Access.
- a `Device` object: a collection of related PVs
- a `Motor` object: a mapping of an Epics Motor
- an `Alarm` object, which can be used to set up notifications when a PV's values goes outside an acceptable bounds.
- an `epics.wx` module that provides wxPython classes designed for use with Epics PVs.

If you're looking to write quick scripts or a simple introduction to using Channel Access, the `caget()` and `caput()` functions are probably where you want to start.

If you're building larger scripts and programs, using `PV` objects provided by the `pv` module is recommended. The `PV` class provides a Process Variable object that has both methods (including `get()` and `put()`) to read and change the PV, and attributes that are kept automatically synchronized with the remote channel. For larger applications, you may find the `Device` class helpful.

The lowest-level CA functionality is exposed in the `ca` module, and companion `dbr` module. While not necessary recommended for most use cases, this module does provide a fairly complete wrapping of the basic EPICS CA library. For people who have used CA from C or other languages, this module should be familiar and seem quite usable, if a little more verbose and C-like than using PV objects.

In addition, the *epics* package contains more specialized modules for alarms, Epics motors, and several other *devices* (collections of PVs), and a set of wxPython widget classes for using EPICS PVs with wxPython.

The *epics* package is targeted for use on Unix-like systems (including Linux and Mac OS X) and Windows with Python versions 2.5, 2.6, 2.7, and 3.1, and 3.2.

2.1 Quick Start

Whether you're familiar with Epics Channel Access or not, start here. You'll then be able to use Python's introspection tools and built-in help system, and the rest of this document as a reference and for detailed discussions.

2.1.1 Functional Approach: caget(), caput()

To get values from PVs, you can use the `caget()` function:

```
>>> from epics import caget, caput
>>> m1 = caget('XXX:m1.VAL')
>>> print m1
1.2001
```

To set PV values, you can use the `caput()` function:

```
>>> caput('XXX:m1.VAL', 1.90)
>>> print caget('XXX:m1.VAL')
1.9000
```

For many cases, this approach is ideal because of its simplicity and clarity.

2.1.2 Object Oriented Approach: PV

If you want to repeatedly access the same PV, you may find it more convenient to ''create a PV object" and use it in a more object-oriented manner.

```
>>> from epics import PV
>>> pv1 = PV('XXX:m1.VAL')
```

PV objects have several methods and attributes. The most important methods are `get()` and `put()` to receive and send the PV's value, and the `value` attribute which stores the current value. In analogy to the `caget()` and `caput()` examples above, the value of a PV can be fetched either with

```
>>> print pv1.get()
1.2001
```

or

```
>>> print pv1.value
1.2001
```

To set a PV's value, you can either use

```
>>> pv1.put(1.9)
```

or assign the `value` attribute

```
>>> pv1.value = 1.9
```

You can see a few of the most important properties of a PV by simply printing it:

```
>>> print pv1
<PV 'XXX:m1.VAL', count=1, type=double, access=read/write>
```

Even more complete information can be seen by printing the PVs `info` attribute:

```
>>> print pv1.info
== XXX:m1.VAL  (native_double) ==
   value      = 1.9
   char_value = '1.90000'
   count      = 1
   nelm       = 1
   type       = double
   units      = mm
   precision  = 5
   host       = somehost.cars.aps.anl.gov:5064
   access     = read/write
   status     = 0
   severity   = 0
   timestamp  = 1265996457.212 (2010-Feb-12 11:40:57.212)
   upper_ctrl_limit    = 12.5
   lower_ctrl_limit    = -12.3
   upper_disp_limit    = 12.5
   lower_disp_limit    = -12.3
   upper_alarm_limit   = 0.0
   lower_alarm_limit   = 0.0
   upper_warning_limit = 0.0
   lower_warning_limit = 0.0
   PV is internally monitored, with 0 user-defined callbacks:
=============================
```

PV objects have several additional methods, especially related to monitoring changes to the PVs and defining functions to be run when the value does change. There are also attributes associated with a PVs *Control Attributes*, like those shown above in the `info` attribute. Further details are at *PV: Epics Process Variables*.

2.2 Functions defined in `epics`: caget(), caput(), etc.

The simplest interface to EPICS Channel Access provides functions `caget()`, `caput()`, as well as functions `camonitor()`, `camonitor_clear()`, and `cainfo()`. These functions are similar to the EPICS command line utilities and to the functions in the EZCA library. They all take the name of an Epics Process Variable (PV) as the first argument. As with the EZCA library, the python implementation keeps an internal cache of connected PV (in this case, using *PV* objects) so that repeated use of a PV name will not actually result in a new connection – see *The get_pv() function and _PVcache_ cache of PVs* for more details. Thus, though the functionality is limited, the performance of the functional approach can be quite good.

2.2.1 `caget()`

`epics.`**`caget`**(*pvname*[, *as_string=False*[, *count=None*[, *as_numpy=True*[, *timeout=None*[, *use_monitor=False*]]]]])

retrieves and returns the value of the named PV.

Parameters

- **`pvname`** – name of Epics Process Variable
- **`as_string`** (`True`/`False`) – whether to return string representation of the PV value.
- **`count`** (integer or `None`) – number of elements to return for array data.
- **`as_numpy`** (`True`/`False`) – whether to return the Numerical Python representation for array data.

- **timeout** (float or `None`) – maximum time to wait (in seconds) for value before returning None.
- **use_monitor** (`True`/`False`) – whether to rely on monitor callbacks or explicitly get value now.

The *count* and *as_numpy* options apply only to array or waveform data. The default behavior is to return the full data array and convert to a numpy array if available. The *count* option can be used to explicitly limit the number of array elements returned, and *as_numpy* can turn on or off conversion to a numpy array.

The *timeout* argument sets the maximum time to wait for a value to be fetched over the network. If the timeout is exceeded, `caget()` will return `None`. This might imply that the PV is not actually available, but it might also mean that the data is large or network slow enough that the data just hasn't been received yet, but may show up later.

The *use_monitor* argument sets whether to rely on the monitors from the underlying PV. The default is `False`, so that each `caget()` will explicitly ask the value to be sent instead of relying on the automatic monitoring normally used for persistent PVs. This makes `caget()` act more like command-line tools, and slightly less efficient than creating a PV and getting values with it. If ultimate performance is a concern, using monitors is recommended. For more details on making `caget()` more efficient, see *Automatic Monitoring of a PV* and *The wait and timeout options for get(), ca.get_complete()*.

The *as_string* argument tells the function to return the **string representation** of the value. The details of the string representation depends on the variable type of the PV. For integer (short or long) and string PVs, the string representation is pretty easy: 0 will become '0', for example. For float and doubles, the internal precision of the PV is used to format the string value. For enum types, the name of the enum state is returned:

```
>>> from epics import caget, caput, cainfo
>>> print caget('XXX:m1.VAL')      # A double PV
0.10000000000000001

>>> print caget('XXX:m1.DESC')     # A string PV
'Motor 1'
>>> print caget('XXX:m1.FOFF')     # An Enum PV
1
```

Adding the *as_string=True* argument always results in string being returned, with the conversion method depending on the data type:

```
>>> print caget('XXX:m1.VAL', as_string=True)
'0.10000'

>>> print caget('XXX:m1.FOFF', as_string=True)
'Frozen'
```

For most array data from Epics waveform records, the regular value will be a numpy array (or a python list if numpy is not installed). The string representation will be something like '<array size=128, type=int>' depending on the size and type of the waveform. An array of doubles might be:

```
>>> print caget('XXX:scan1.P1PA')   # A Double Waveform
array([-0.08      , -0.078     , -0.076     , ...,
    1.99599814, 1.99799919,  2.      ])

>>> print caget('XXX:scan1.P1PA', as_string=True)
'<array size=2000, type=DOUBLE>'
```

As an important special case, CHAR waveforms will be turned to Python strings when *as_string* is `True`. This is useful to work around the low limit of the maximum length (40 characters!) of EPICS strings, and means that it is fairly common to use CHAR waveforms when long strings are desired:

```
>>> print caget('XXX:dir')       # A CHAR waveform
array([ 84,  58,  92, 120,  97, 115,  95, 117, 115,
   101, 114,  92,  77,  97, 114,  99, 104,  50,  48,
    49,  48,  92,  70,  97, 115, 116,  77,  97, 112])

>>> print caget('XXX:dir',as_string=True)
'T:\\xas_user\\March2010\\Fastmap'
```

Of course,character waveforms are not always used for long strings, but can also hold byte array data, such as comes from some detectors and devices.

2.2.2 `caput()`

`epics.caput` (*pvname*, *value*[, *wait=False*[, *timeout=60*]])
set the value of the named PV.

Parameters

- **pvname** – name of Epics Process Variable
- **value** – value to send.
- **wait** (`True`/`False`) – whether to wait until the processing has completed.
- **timeout** (*double*) – how long to wait (in seconds) for put to complete before giving up.

Return type integer

The optional *wait* argument tells the function to wait until the processing completes. This can be useful for PVs which take significant time to complete, either because it causes a physical device (motor, valve, etc) to move or because it triggers a complex calculation or data processing sequence. The *timeout* argument gives the maximum time to wait, in seconds. The function will return after this (approximate) time even if the `caput()` has not completed.

This function returns 1 on success, and a negative number if the timeout has been exceeded.

```
>>> from epics import caget, caput, cainfo
>>> caput('XXX:m1.VAL',2.30)
1
>>> caput('XXX:m1.VAL',-2.30, wait=True)
... waits a few seconds ...
1
```

2.2.3 `cainfo()`

`epics.cainfo` (*pvname*[, *print_out=True*])
prints (or returns as a string) an informational paragraph about the PV, including Control Settings.

Parameters

- **pvname** – name of Epics Process Variable
- **print_out** – whether to write results to standard output (otherwise the string is returned).

2.2.4 `camonitor()`

`epics.camonitor` (*pvname*[, *writer=None*[, *callback=None*]])
This *sets a monitor* on the named PV, which will cause *something* to be done each time the value changes. By

default the PV name, time, and value will be printed out (to standard output) when the value changes, but the action that actually happens can be customized.

Parameters

- **pvname** – name of Epics Process Variable
- **writer** (None or a callable function that takes a string argument.) – where to write results to standard output .
- **callback** (None or callable function) – user-supplied function to receive result

One can specify any function that can take a string as *writer*, such as the `write()` method of an open file that has been open for writing. If left as `None`, messages of changes will be sent to `sys.stdout.write()`. For more complete control, one can specify a *callback* function to be called on each change event. This callback should take keyword arguments for *pvname*, *value*, and *char_value*. See *User-supplied Callback functions* for information on writing callback functions for `camonitor()`.

```
>>> from epics import camonitor
>>> camonitor('XXX.m1.VAL')
XXX.m1.VAL 2010-08-01 10:34:15.822452 1.3
XXX.m1.VAL 2010-08-01 10:34:16.823233 1.2
XXX.m1.VAL 2010-08-01 10:34:17.823233 1.1
XXX.m1.VAL 2010-08-01 10:34:18.823233 1.0
```

2.2.5 camonitor_clear()

epics.**camonitor_clear**(*pvname*)

clears a monitor set on the named PV by `camonitor()`.

Parameters **pvname** – name of Epics Process Variable

This simple example monitors a PV with `camonitor()` for while, with changes being saved to a log file. After a while, the monitor is cleared and the log file is inspected:

```
>>> import epics
>>> fh = open('PV1.log','w')
>>> epics.camonitor('XXX:DMM1Ch2_calc.VAL',writer=fh.write)
>>> .... wait for changes ...
>>> epics.camonitor_clear('XXX:DMM1Ch2_calc.VAL')
>>> fh.close()
>>> fh = open('PV1.log','r')
>>> for i in fh.readlines(): print i[:-1]
 XXX:DMM1Ch2_calc.VAL 2010-03-24 11:56:40.536946 -183.5035
 XXX:DMM1Ch2_calc.VAL 2010-03-24 11:56:41.536757 -183.6716
 XXX:DMM1Ch2_calc.VAL 2010-03-24 11:56:42.535568 -183.5112
 XXX:DMM1Ch2_calc.VAL 2010-03-24 11:56:43.535379 -183.5466
 XXX:DMM1Ch2_calc.VAL 2010-03-24 11:56:44.535191 -183.4890
 XXX:DMM1Ch2_calc.VAL 2010-03-24 11:56:45.535001 -183.5066
 XXX:DMM1Ch2_calc.VAL 2010-03-24 11:56:46.535813 -183.5085
 XXX:DMM1Ch2_calc.VAL 2010-03-24 11:56:47.536623 -183.5223
 XXX:DMM1Ch2_calc.VAL 2010-03-24 11:56:48.536434 -183.6832
```

2.3 Motivation: Why another Python-Epics Interface?

PyEpics version 3 is intended as an improvement over EpicsCA 2.1, and should replace that older Epics-Python interface. That version had performance issues, especially when connecting to a large number of PVs, is not thread-aware, and has become difficult to maintain for Windows and Linux.

There are a few other Python modules exposing Epics Channel Access available. Most of these have a interface to the CA library that was both closer to the C library and lower-level than EpicsCA. Most of these interfaces use specialized C-Python ‘wrapper’ code to provide the interface.

Because of this, an additional motivation for this package was to allow a more common interface to be used that built higher-level objects (as EpicsCA had) on top of a complete lower-level interface. The desire to come to a more universally-acceptable Python-Epics interface has definitely influenced the goals for this module, which include:

1. providing both low-level (C-like) and higher-level access (Pythonic objects) to the EPICS Channel Access protocol.
2. supporting as many features of Epics 3.14 as possible, including preemptive callbacks and thread support.
3. easy support and distribution for Windows and Unix-like systems.
4. being ready for porting to Python3.
5. using Python’s ctypes library.

The main implementation feature used here (and difference from EpicsCA) is using Python’s ctypes library to handle the connection between Python and the CA C library. Using ctypes has many advantages. Principally, it fully eliminates the need to write (and maintain) wrapper code either with SWIG or directly with Python’s C API. Since the ctypes module allows access to C data and objects in pure Python, no compilation step is needed to build the module, making installation and support on multiple platforms much easier. Since ctypes loads a shared object library at runtime, the underlying Epics Channel Access library can be upgraded without having to re-build the Python wrapper. In addition, using ctypes provides the most reliable thread-safety available, as each call to the underlying C library is automatically made thread-aware without explicit code. Finally, by avoiding the C API altogether, migration to Python3 is greatly simplified. PyEpics3 does work with both Python 2.* and 3.*.

2.4 Status and To-Do List

The PyEpics package is actively maintained, but the core library seems reasonably stable and ready to use in production code. Features are being added slowly, and testing is integrated into development so that the changce of introducing bugs into existing codes is minimized. The package is targeted and tested to work with Python 2.6, 2.7, 3.2 and 3.3 simultaneously (that is, the same code is meant to support all these versions).

There are several desired features are left unfinished:

- add more Epics Devices, including low-level epics records and more suport for Area Detectors.
- incorporate some or all of the Channel Access Server from pcaspy
- build and improve applications using PyEpics, especially for common data acquistion needs.
- improve and extend the use of PyQt widgets with PyEpics.

If you are interested in working on any of these or other topics, please contact the authors.

CHAPTER

THREE

PV: EPICS PROCESS VARIABLES

The *pv* module provides a higher-level class *pv.PV*, which creates a *PV* object for an EPICS Process Variable. A *PV* object has both methods and attributes for accessing it's properties.

3.1 The PV class

class pv.**PV**(*pvname*[, *callback=None*[, *form='time'*[, *auto_monitor=None*[, *connection_callback=None*[, *connection_timeout=None*[, *verbose=False*]]]]]])

create a PV object for a named Epics Process Variable.

Parameters

- **pvname** – name of Epics Process Variable
- **callback** (*callable, tuple, list or None*) – user-defined callback function on changes to PV value or state.
- **form** (*string, one of ('native','ctrl', or 'time')*) – which epics *data type* to use: the 'native', 'time', or the 'ctrl' (Control) variant.
- **auto_monitor** (None, True, False, or bitmask (see *Automatic Monitoring of a PV*)) – whether to automatically monitor the PV for changes.
- **connection_callback** (callable or None) – user-defined function called on changes to PV connection status.
- **connection_timeout** (float or None) – time (in seconds) to wait for connection before giving up
- **verbose** (True/False) – whether to print out debugging messages

Once created, a PV should (barring any network issues) automatically connect and be ready to use.

```
>>> from epics import PV
>>> p = PV('XX:m1.VAL')
>>> print p.get()
>>> print p.count, p.type
```

The *pvname* is required, and is the name of an existing Process Variable.

The *callback* parameter specifies one or more python methods to be called on changes, as discussed in more detail at *User-supplied Callback functions*

The *connection_callback* parameter specifies a python method to be called on changes to the connection status of the PV (that is, when it connects or disconnects). This is discussed in more detail at *User-supplied Connection Callback functions*

The *form* parameter specifies which of the three variants 'native', 'ctrl' (Control) or 'time' (the default) to use for the PV. The 'native' form returns just the value, the 'time' form includes the timestamp from the server the PV lives on, as well as status information. The control form includes several additional fields such as limits to the PV, which can be useful in some cases. Also note that the additional 'ctrl' value fields (see the *Table of Control Attributes*) can be obtained with `get_ctrlvars()` even for PVs of 'native' or 'time' form.

The *auto_monitor* parameter specifies whether the PV should be automatically monitored. See *Automatic Monitoring of a PV* for a detailed description of this.

The *verbose* parameter specifies more verbose output on changes, and is intended for debugging purposes.

3.1.1 methods

A *PV* has several methods for getting and setting its value and defining callbacks to be executed when the PV changes.

`pv.`**`get`**`(`[*count=None*[, *as_string=False*[, *as_numpy=True*[, *timeout=None*[, *use_monitor=True*]]]]]`)`

get and return the current value of the PV

Parameters

- **`count`** (integer or `None`) – maximum number of array elements to return
- **`as_string`** (`True`/`False`) – whether to return the string representation of the value.
- **`as_numpy`** – whether to try to return a numpy array where appropriate.
- **`timeout`** (float or `None`) – maximum time to wait for data before returning `None`.
- **`use_monitor`** (`True`/`False`) – whether to rely on monitor callbacks or explicitly get value now.

see *String representation for a PV* for details on how the string representation is determined.

With the *as_numpy* option, an array PV (that is, a PV whose value has more than one element) will be returned as a numpy array, provided the numpy module is available. See *Strategies for working with large arrays* for a discussion of strategies for how to best deal with very large arrays.

The *use_monitor* option controls whether the most recent value from the automatic monitoring will be used or whether the value will be explicitly asked for right now. Usually, you can rely on a PVs value being kept up to date, and so the default here is `True`. But, since network traffic is not instantaneous and hard to predict, the value returned with *use_monitor=True* may be out-of-date.

The *timeout* sets how long (in seconds) to wait for the value to be sent. This only applies with *use_monitor=False*, or if the PV is not automatically monitored. Otherwise, the most recently received value will be sent immediately.

See *Automatic Monitoring of a PV* for more on monitoring PVs and *The wait and timeout options for get(), ca.get_complete()* for more details on what happens when a `pv.get()` times out.

`pv.`**`put`**`(`*value*[, *wait=False*[, *timeout=30.0*[, *use_complete=False*[, *callback=None*[, *callback_data=None*]]]]]`)`

set the PV value, optionally waiting to return until processing has completed, or setting the `put_complete` to indicate complete-ness.

Parameters

- **`value`** – value to set PV
- **`wait`** (`True`/`False`) – whether to wait for processing to complete (or time-out) before returning.

- **`timeout`** (*float*) – maximum time to wait for processing to complete before returning anyway.
- **`use_complete`** (`True`/`False`) – whether to use a built-in callback to set *`put_complete`*.
- **`callback`** (`None` or a valid python function) – user-supplied function to run when processing has completed.
- **`callback_data`** – extra data to pass on to a user-supplied callback function.

The *wait* and *callback* arguments, as well as the 'use_complete' / *`put_complete`* attribute give a few options for knowing that a *`put()`* has completed. See *Put with wait, put callbacks, and put_complete* for more details.

`pv.`**`get_ctrlvars`**`()`

returns a dictionary of the **control values** for the PV. This dictionary may have many members, depending on the data type of PV. See the *Table of Control Attributes* for details.

`pv.`**`poll`**`(`[*evt=1.e-4*[, *iot=1.0*]]`)`

poll for changes. This simply calls `ca.poll()`

Parameters

- **`evt`** (*float*) – time to pass to `ca.pend_event()`
- **`iot`** (*float*) – time to pass to `ca.pend_io()`

`pv.`**`connect`**`(`[*timeout=None*]`)`

this explicitly connects a PV, and returns whether or not it has successfully connected. It is probably not that useful, as connection should happen automatically. See *`wait_for_connection()`*.

Parameters **`timeout`** (*float*) – maximum connection time, passed to `ca.connect_channel()`

Return type `True`/`False`

if timeout is `None`, the PVs connection_timeout parameter will be used. If that is also `None`, `ca.DEFAULT_CONNECTION_TIMEOUT` will be used.

`pv.`**`wait_for_connection`**`(`[*timeout=None*]`)`

this waits until a PV is connected, or has timed-out waiting for a connection. Returns whether the connection has occurred.

Parameters **`timeout`** (*float*) – maximum connection time.

Return type `True`/`False`

if timeout is `None`, the PVs connection_timeout parameter will be used. If that is also `None`, `ca.DEFAULT_CONNECTION_TIMEOUT` will be used.

`pv.`**`disconnect`**`()`

disconnect a PV, clearing all callbacks.

`pv.`**`add_callback`**`(`*callback=None[, index=None [, with_ctrlvars=True[, **kw]]*`)`

adds a user-defined callback routine to be run on each change event for this PV. Returns the integer *index* for the callback.

Parameters

- **`callback`** (`None` or callable) – user-supplied function to run when PV changes.
- **`index`** (`None` (integer will be produced) or immutable) – identifying key for this callback
- **`with_ctrlvars`** – whether to (try to) make sure that accurate `control values` will be sent to the callback.
- **`kw`** – additional keyword/value arguments to pass to each execution of the callback.

Return type integer

Note that multiple callbacks can be defined, each having its own index (a dictionary key, typically an integer). When a PV changes, all the defined callbacks will be executed. They will be called in order (by sorting the keys of the `callbacks` dictionary)

See also: `callbacks` attribute, *User-supplied Callback functions*

`pv.`**`remove_callback`**(*index=None*)
remove a user-defined callback routine using supplied

Parameters **`index`** (`None` or integer) – index of user-supplied function, as returned by `add_callback()`, and also to key for this callback in the `callbacks` dictionary.

Return type integer

If only one callback is defined an index=‘‘None‘‘, this will clear the only defined callback.

See also: `callbacks` attribute, *User-supplied Callback functions*

`pv.`**`clear_callbacks`**()
remove all user-defined callback routine.

`pv.`**`run_callbacks`**()
execute all user-defined callbacks right now, even if the PV has not changed. Useful for debugging!

See also: `callbacks` attribute, *User-supplied Callback functions*

`pv.`**`run_callback`**(*index*)
execute a particular user-defined callback right now, even if the PV has not changed. Useful for debugging!

See also: `callbacks` attribute, *User-supplied Callback functions*

3.1.2 attributes

A PV object has many attributes, each associated with some property of the underlying PV: its *value*, *host*, *count*, and so on. For properties that can change, the PV attribute will hold the latest value for the corresponding property, Most attributes are **read-only**, and cannot be assigned to. The exception to this rule is the `value` attribute.

`pv.`**`value`**
The current value of the PV.

Note: The `value` attribute can be assigned to. When read, the latest value will be returned, even if that means a `get()` needs to be called.

Assigning to `value` is equivalent to setting the value with the `put()` method.

```
>>> from epics import PV
>>> p1 = PV('xxx.VAL')
>>> print p1.value
1.00
>>> p1.value = 2.00
```

`pv.`**`char_value`**
The string representation of the string, as described in `get()`.

`pv.`**`status`**
The PV status, which will be 1 for a Normal, connected PV.

`pv.`**`type`**
string describing data type of PV, such as *double*, *float*, *enum*, *string*, *int*, *long*, *char*, or one of the *ctrl* or *time* variants of these, which will be named *ctrl_double*, *time_enum*, and so on. See the *Table of DBR Types*

pv.**ftype**
The integer value (from the underlying C library) indicating the PV data type according to *Table of DBR Types*

pv.**host**
string of host machine provide this PV.

pv.**count**
number of data elements in a PV. 1 except for waveform PVs, where it gives the number of elements in the waveform. For recent versions of Epics Base (3.14.11 and later?), this gives the *.NORD* field, which gives the number of elements last put into the PV and which may be less than the maximum number allowed (see *nelm* below).

pv.**nelm**
number of data elements in a PV. 1 except for waveform PVs where it gives the maximum number of elements in the waveform. For recent versions of Epics Base (3.14.11 and later?), this gives the *.NELM* parameter. See also the *count* attribute above.

pv.**read_access**
Boolean (`True`/`False`) for whether PV is readable

pv.**write_access**
Boolean (`True`/`False`) for whether PV is writable

pv.**access**
string describing read/write access. One of 'read/write','read-only','write-only', 'no access'.

pv.**severity**
severity value of PV. Usually 0 for PVs that are not in an alarm condition.

pv.**timestamp**
Unix (not Epics!!) timestamp of the last seen event for this PV. Note that this is will contain the timestamp from the Epics record if the PV object was created with the `form='time'` option. Otherwise, the timestamp will be the timestamp according to the client, indicating when the data arrive from the server.

pv.**precision**
number of decimal places of precision to use for float and double PVs

pv.**units**
string of engineering units for PV

pv.**enum_strs**
a list of strings for the enumeration states of this PV (for enum PVs)

pv.**info**
a string paragraph (ie, including newlines) showing much of the information about the PV.

pv.**upper_disp_limit**

pv.**lower_disp_limit**

pv.**upper_alarm_limit**

pv.**lower_alarm_limit**

pv.**lower_warning_limit**

pv.**upper_warning_limit**

pv.**upper_ctrl_limit**

pv.**lower_ctrl_limit**
These are all the various kinds of limits for a PV.

pv.**put_complete**
a Boolean (`True`/`False`) value for whether the most recent `put()` has completed.

pv.**callbacks**
a dictionary of currently defined callbacks, to be run on changes to the PV. This dictionary has integer keys (generally in increasing order of when they were defined) which sets which order for executing the callbacks. The values of this dictionary are tuples of *(callback, keyword_arguments)*.

Note: The `callbacks` attribute can be assigned to or manipulated directly. This is not recommended. Use the methods `add_callback()`, `remove_callback()`, and `clear_callbacks()` instead of altering this dictionary directly.

pv.**connection_callbacks**
a simple list of connection callbacks: functions to be run when the connection status of the PV changes. See *User-supplied Connection Callback functions* for more details.

3.2 String representation for a PV

The string representation for a *PV*, as returned either with the *as_string* argument to `ca.get()` or from the `char_value` attribute (they are equivalent) needs some further explanation.

The value of the string representation (hereafter, the `char_value`), will depend on the native type and count of a *PV*. *Table of String Representations*

Table of String Representations: How raw data `value` is mapped to `char_value` for different native data types.

data types	*count*	*char_value*
string	1	= value
char	1	= value
short	1	= str(value)
long	1	= str(value)
enum	1	= enum_str[value]
double	1	= (“%%.%if” % (precision)) % value
float	1	= (“%%.%if” % (precision)) % value
char	> 1	= long string from bytes in array
all others	> 1	= <array size=*count*, type=*type*>

For double/float values with large exponents, the formatting will be *(“%%.%ig” % (precision)) % value*. For character waveforms (*char* data with *count* > 1), the `char_value` will be set according to:

```
>>> firstnull  = val.index(0)
>>> if firstnull == -1: firstnull= len(val)
>>> char_value = ''.join([chr(i) for i in val[:firstnull]).rstrip()
```

3.3 Automatic Monitoring of a PV

When creating a PV, the *auto_monitor* parameter specifies whether the PV should be automatically monitored or not. Automatic monitoring means that an internal callback will be registered for changes. Any callbacks defined by the user will be called by this internal callback when changes occur.

For most scalar-value PVs, this automatic monitoring is desirable, as the PV will see all changes (and run callbacks) without any additional interaction from the user. The PV’s value will always be up-to-date and no unnecessary network traffic is needed.

Possible values for `auto_monitor` are:

`False` For some PVs, especially those that change much more rapidly than you care about or those that contain large arrays as values, auto_monitoring can add network traffic that you don't need. For these, you may wish to create your PVs with *auto_monitor=False*. When you do this, you will need to make calls to `get()` to explicitly get the latest value.

`None` The default value for *auto_monitor* is `None`, and is set to `True` if the element count for the PV is smaller than `ca.AUTOMONITOR_MAXLENGTH` (default of 65536). To suppress monitoring of PVs with fewer array values, you will have to explicitly turn *auto_monitor* to `False`. For waveform arrays with more elements, automatic monitoring will not be done unless you explicitly set *auto_monitor=True*, or to an explicit mask. See *Strategies for working with large arrays* for more details.

`True` When *auto_monitor* is set to `True`, the value will be monitored using the default subscription mask set at `ca.DEFAULT_SUBSCRIPTION_MASK`.

This mask determines which kinds of changes cause the PV to update. By default, the subscription updates when the PV value changes by more than the monitor deadband, or when the PV alarm status changes. This behavior is the same as the default in EPICS' *camonitor* tool.

Mask It is also possible to request an explicit type of CA subscription by setting *auto_monitor* to a numeric subscription mask made up of dbr.DBE_ALARM, dbr.DBE_LOG and/or dbr.DBE_VALUE. This mask will be passed directly to `ca.create_subscription()` An example would be:

```
pv1 = PV('AAA', auto_monitor=dbr.DBE_VALUE)
pv2 = PV('BBB', auto_monitor=dbr.DBE_VALUE|dbr.DBE_ALARM)
pv3 = PV('CCC', auto_monitor=dbr.DBE_VALUE|dbr.DBE_ALARM|dbr.DBE_LOG)
```

which will generate callbacks for pv1 only when the value of 'AAA' changes, while pv2 will receive callbacks if the value or alarm state of 'BBB' changes, and pv3 will receive callbacks for all changes to 'CCC'. Note that these dbr.DBE_**** constants are ORed together as a bitmask.

3.4 User-supplied Callback functions

This section describes user-defined functions that are called when the value of a PV changes. These callback functions are useful as they allow you to be notified of changes without having to continually ask for a PVs current value. Much of this information is similar to that in *User-supplied Callback functions* for the `ca` module, though there are some important enhancements to callbacks on *PV* objects.

You can define more than one callback function per PV to be run on value changes. These functions can be specified when creating a PV, with the *callback* argument which can take either a single callback function or a list or tuple of callback functions. After a PV has been created, you can add callback functions with `add_callback()`, remove them with `remove_callback()`, and explicitly run them with `run_callback()`. Each callback has an internal unique *index* (a small integer number) that can be used for specifying which one to add, remove, and run.

When defining a callback function to be run on changes to a PV, it is important to know two things:

1. how your function will be called.
2. what is permissible to do inside your callback function.

Callback functions will be called with several keyword arguments. You should be prepared to have them passed to your function, and should always include ***kw* to catch all arguments. Your callback will be sent the following keyword parameters:

- *pvname*: the name of the pv
- *value*: the latest value

- *char_value*: string representation of value
- *count*: the number of data elements
- *ftype*: the numerical CA type indicating the data type
- *type*: the python type for the data
- *status*: the status of the PV (1 for OK)
- *precision*: number of decimal places of precision for floating point values
- *units*: string for PV units
- *severity*: PV severity
- *timestamp*: timestamp from CA server.
- *read_access*: read access (`True`/`False`)
- *write_access*: write access (`True`/`False`)
- *access*: string description of read- and write-access
- *host*: host machine and CA port serving PV
- *enum_strs*: the list of enumeration strings
- *upper_disp_limit*: upper display limit
- *lower_disp_limit*: lower display limit
- *upper_alarm_limit*: upper alarm limit
- *lower_alarm_limit*: lower alarm limit
- *upper_warning_limit*: upper warning limit
- *lower_warning_limit*: lower warning limit
- *upper_ctrl_limit*: upper control limit
- *lower_ctrl_limit*: lower control limit
- *chid*: integer channel ID
- ***cb_info*: (index, self) tuple containing callback ID** and the PV object

Some of these may not be directly applicable to all PV data types, and some values may be `None` if the control parameters have not yet been fetched with `get_ctrlvars()`.

It is important to keep in mind that the callback function will be run *inside* a CA function, and cannot reliably make any other CA calls. It is helpful to think "this all happens inside of a `pend_event()` call", and in an epics thread that may or may not be the main thread of your program. It is advisable to keep the callback functions short and not resource-intensive. Consider strategies which use the callback only to record that a change has occurred and then act on that change later – perhaps in a separate thread, perhaps after `pend_event()` has completed.

The *cb_info* parameter supplied to the callback needs special attention, as it is the only non-Epics information passed. The *cb_info* parameter will be a tuple containing (`index`, `self`) where `index` is the key for the `callbacks` dictionary for the PV and `self` *is* PV object. A principle use of this tuple is to **remove the current callback** if an error happens, as for example in GUI code if the widget that the callback is meant to update disappears.

3.5 User-supplied Connection Callback functions

A *connection* callback is a user-defined function that is called when the connection status of a PV changes – that is, when a PV initially connects, disconnects or reconnects due to the process serving the PV going away, or loss of network connection. A connection callback can be specified when a PV is created, or can be added by appending to the `connection_callbacks` list. If there is more than one connection callback defined, they will all be run when the connection state changes.

A connection callback should be prepared to receive the following keyword arguments:

- *pvname*: the name of the pv
- *conn*: the connection status

where *conn* will be either `True` or `` `False``, specifying whether the PV is now connected. A simple example is given below.

3.6 Put with wait, put callbacks, and put_complete

Some EPICS records take a significant amount of time to fully process, and sometimes you want to wait until the processing completes before going on. There are a few ways to accomplish this. First, one can simply wait until the processing is done:

```
import epics
p = epics.PV('XXX')
p.put(1.0, wait=True)
print 'Done'
```

This will hang until the processing of the PV completes (motor moving, etc) before printing 'Done'. You can also specify a maximum time to wait – a *timeout* (in seconds):

```
p.put(1.0, wait=True, timeout=30)
```

which will wait up to 30 seconds. For the pedantic, this timeout should not be used as an accurate clock – the actual wait time may be slightly longer.

A second method is to use the 'use_complete' option and watch for the `put_complete` attribute to become `True` after a `put()`. This is somewhat more flexible than using *wait=True* as above, because you can more carefully control how often you look for a `put()` to complete, and what to do in the interim. A simple example would be:

```
p.put(1.0, use_complete=True)
waiting = True
while waiting:
    time.sleep(0.001)
    waiting = not p.put_complete
```

An additional advantage of this approach is that you can easily wait for multiple PVs to complete with python's built-in *all* function, as with:

```
pvgroup = (epics.PV('XXX'), epics.PV('YYY'), epics.PV('ZZZ'))
newvals = (1.0, 2.0,  3.0)
for pv, val in zip(pvgroup, newvals):
    pv.put(val, use_complete=True)

waiting = True
while waiting:
    time.sleep(0.001)
    waiting = not all([pv.put_complete for pv in pvgroup])
print 'All puts are done!'
```

For maximum flexibility, one can all define a *put callback*, a function to be run when the `put()` has completed. This function requires a *pvname* keyword argument, but will receive no others, unless you pass in data with the *callback_data* argument (which should be dict-like) to `put()`. A simple example would be:

```
pv = epics.PV('XXX')
def onPutComplete(pvname=None, **kws):
    print 'Put done for %s' % pvname

pv.put(1.0, callback=onPutComplete)
```

3.7 The `get_pv()` function and `_PVcache_` cache of PVs

As mentioned in the previous chapter, a cache of PVs is maintained for each process using pyepics. When using `epics.caget()`, `epics.caput()` and so forth, or when creating a `PV` directly, the corresponding PV is kept in a global cache, held in `pv._PVcache_`.

The function `get_pv()` will retrieve the named PV from this cache, or create a new `PV` if one is not found. In long-running or complex processes, it is not unusual to access a particular PV many times, perhaps calling a function that creates a PV but only keeping that PV object for the life of the function. Using `get_pv()` instead of creating a `PV` can improve performance (the PV is already connected) and is highly recommended.

pv.**get_pv**(*pvname*[, *form='time'*[, *connect=False*[, *timeout=5*[, *context=None*[, ***kws*]]]]])
retrieves a PV from `_PVcache` or creates and returns a new PV.

Parameters

- **pvname** – name of Epics Process Variable
- **form** (*string, one of ('native','ctrl', or 'time')*) – which epics *data type* to use: the 'native' , or the 'ctrl' (Control) or 'time' variant.
- **connect** (`True`/`False`) – whether to wait for the PV to connect.
- **timeout** (float or `None`) – maximum time to wait (in seconds) for value before returning None.
- **context** (integer or `None` (default)) – integer threading context.

Additional keywords are passed directly to `PV`.

pv.**_PVcache_**
A cache of `PV` objects for the process.

3.8 Examples

Some simple examples using PVs follow.

3.8.1 Basic Use

The simplest approach is to simply create a PV and use its `value` attribute:

```
>>> from epics import PV
>>> p1 = PV('xxx.VAL')
>>> print p1.value
1.00
>>> p1.value = 2.00
```

The *print p1.value* line automatically fetches the current PV value. The *p1.value = 2.00* line does a `put()` to set the value, causing any necessary processing over the network.

The above example is equivalent to

```
>>> from epics import PV
>>> p1 = PV('xxx.VAL')
>>> print p1.get()
1.00
>>> p1.put(value = 2.00)
```

To get a string representation of the value, you can use either

```
>>> print p1.get(as_string=True)
'1.000'
```

or, equivalently

```
>>> print p1.char_value
'1.000'
```

3.8.2 Example of using info and more properties examples

A PV has many attributes. This can be seen from its *info* paragraph:

```
>>> import epics
>>> p = epics.PV('13IDA:m3')
>>> print p.info
== 13IDA:m3  (native_double) ==
   value      = 0.2
   char_value = '0.200'
   count      = 1
   type       = double
   units      = mm
   precision  = 3
   host       = ioc13ida.cars.aps.anl.gov:5064
   access     = read/write
   status     = 0
   severity   = 0
   timestamp  = 1274809682.967 (2010-05-25 12:48:02.967364)
   upper_ctrl_limit    = 5.49393415451
   lower_ctrl_limit    = -14.5060658455
   upper_disp_limit    = 5.49393415451
   lower_disp_limit    = -14.5060658455
   upper_alarm_limit   = 0.0
   lower_alarm_limit   = 0.0
   upper_warning_limit = 0.0
   lower_warning_limit = 0.0
   PV is internally monitored, with 0 user-defined callbacks:
=============================
```

The individual attributes can also be accessed as below. Many of these (the *control attributes*, see *Table of Control Attributes*) will not be filled in until either the `info` attribute is accessed or until `get_ctrlvars()` is called.

```
>>>  print p.type
double
>>> print p.units, p.precision, p.lower_disp_limit
mm 3 -14.5060658455
```

3.8.3 Getting a string value

It is not uncommon to want a string representation of a PVs value, for example to show in a display window or to write to some report. For string PVs and integer PVs, this is a simple task. For floating point values, there is ambiguity how many significant digits to show. EPICS PVs all have a `precision` field. which sets how many digits after the decimal place should be described. In addition, for ENUM PVs, it would be desire able to get at the name of the ENUM state, not just its integer value.

To get the string representation of a PVs value, use either the `char_value` attribute or the *as_string=True* argument to `get()`

3.8.4 Example of `put()`

To put a new value to a variable, either of these two approaches can be used:

```
>>> import epics
>>> p = epics.PV('XXX')
>>> p.put(1.0)
```

Or (equivalently):

```
>>> import epics
>>> p = epics.PV('XXX')
>>> p.value = 1.0
```

The `value` attribute is the only attribute that can be set.

3.8.5 Example of simple callback

It is often useful to get a notification of when a PV changes. In general, it would be inconvenient (and possibly inefficient) to have to continually ask if a PVs value has changed. Instead, it is better to set a *callback* function: a function to be run when the value has changed.

A simple example of this would be:

```
import epics
import time
def onChanges(pvname=None, value=None, char_value=None, **kw):
    print 'PV Changed! ', pvname, char_value, time.ctime()


mypv = epics.PV(pvname)
mypv.add_callback(onChanges)

print 'Now wait for changes'

t0 = time.time()
while time.time() - t0 < 60.0:
    time.sleep(1.e-3)
print 'Done.'
```

This first defines a *callback function* called *onChanges()* and then simply waits for changes to happen. Note that the callback function should take keyword arguments, and generally use ***kw* to catch all arguments. See *User-supplied Callback functions* for more details.

3.8.6 Example of connection callback

A connection callback:

```
#
# example of using a connection callback that will be called
# for any change in connection status

import epics
import time
import sys
from  pvnames import motor1

write = sys.stdout.write
def onConnectionChange(pvname=None, conn= None, **kws):
    write('PV connection status changed: %s %s\n' % (pvname,  repr(conn)))
    sys.stdout.flush()

def onValueChange(pvname=None, value=None, host=None, **kws):
    write('PV value changed: %s (%s)  %s\n' % ( pvname, host, repr(value)))
    sys.stdout.flush()
mypv = epics.PV(motor1,
                connection_callback= onConnectionChange,
                callback= onValueChange)

mypv.get()

write('Now waiting, watching values and connection changes:\n')
t0 = time.time()
while time.time()-t0 < 300:
    time.sleep(0.01)
```

CHAPTER

FOUR

CA: LOW-LEVEL CHANNEL ACCESS MODULE

The `ca` module provides a low-level wrapping of the EPICS Channel Access (CA) library, using ctypes. Most users of the *epics* module will not need to be concerned with most of the details here, and will instead use the simple functional interface (`epics.caget()`, `epics.caput()` and so on), or use the `epics.PV` class to create and use epics PV objects.

4.1 General description, difference with C library

The goal of the `ca` module is to provide a fairly complete mapping of the C interface to the CA library while also providing a pleasant Python experience. It is expected that anyone looking into the details of this module is somewhat familiar with Channel Access and knows where to consult the Channel Access Reference Documentation. This document focuses on the differences with the C interface, assuming a general understanding of what the functions are meant to do.

4.1.1 Name Mangling

As a general rule, a CA function named *ca_XXX* in the C library will have the equivalent function called *XXX* in the *ca* module. This is because the intention is that one will import the *ca* module with

```
>>> from epics import ca
```

so that the Python function `ca.XXX()` will corresponds to the C function *ca_XXX*. That is, the CA library called its functions *ca_XXX* because C does not have namespaces. Python does have namespaces, and so they are used.

Similar name *un-mangling* also happens with the DBR prefixes for constants, held here in the *dbr* module. Thus, the C constant DBR_STRING becomes dbr.STRING in Python.

4.1.2 Other Changes and Omissions

Several function in the C version of the CA library are not implemented in the Python module. Most of these unimplemented functions are currently seen as unnecessary for Python, though some of these could be added without much trouble if needed. See *Omissions* for further details.

In addition, while the CA library supports several *DBR* types in C, not all of these are supported in Python. Only native types and their DBR_TIME and DBR_CTRL variants are supported here. The DBR_STS and DBR_GR variants are not, as they are subsets of the DBR_CTRL type, and space optimization is not something you'll be striving for with Python. Several *dbr_XXX* functions are also not supported, as they appear to be needed only to be able to dynamically allocate memory, which is not necessary in Python.

4.2 Initialization, Finalization, and Life-cycle

The Channel Access library must be initialized before it can be used. There are 3 main reasons for this need:

1. CA requires a context model (preemptive callbacks or non-preemptive callbacks) to be specified before any actual calls can be made.

2. the ctypes interface requires that the shared library be loaded before it is used.

3. ctypes also requires that references to the library and callback functions be kept for the life-cycle of CA-using part of a program (or else they will be garbage collected).

As far as is possible, the `ca` module hides the details of the CA lifecyle from the user, so that it is not necessary to to worry about explicitly initializing a Channel Access session. Instead, the library is initialized as soon as it is needed, and intervention is really only required to change default settings. The `ca` module also handles finalizing the CA session, so that core-dumps and warning messages do not happen due to CA still being 'alive' as a program ends.

Because some users may wish to customize the initialization and finalization process, the detailed steps will be described here. These initialization and finalization tasks are handled in the following way:

- The `libca` variable in the `ca` module holds a permanent, global reference to the CA shared object library (DLL).
- the function `initialize_libca()` is called to initialize libca. This function takes no arguments, but does use the global Boolean `PREEMPTIVE_CALLBACK` (default value of `True`) to control whether preemptive callbacks are used.
- the function `finalize_libca()` is used to finalize libca. Normally, this is function is registered to be called when a program ends with `atexit.register()`. Note that this only gets called on a graceful shutdown. If the program crashes (for a non-CA related reason, for example), this finalization may not be done, and connections to Epics Variables may not be closed completely on the Channel Access server.

`epics.ca.`**`PREEMPTIVE_CALLBACK`**

sets whether preemptive callbacks will be used. The default value is `True`. If you wish to run without preemptive callbacks this variable *MUST* be set before any other use of the CA library. With preemptive callbacks enabled, EPICS communication will not require client code to continually poll for changes. With preemptive callback disables, you will need to frequently poll epics with `pend_io()` and func:*pend_event*.

`epics.ca.`**`DEFAULT_CONNECTION_TIMEOUT`**

sets the default *timeout* value (in seconds) for `connect_channel()`. The default value is *2.0*

`epics.ca.`**`AUTOMONITOR_MAXLENGTH`**

sets the default array length (ie, how many elements an array has) above which automatic conversion to numpy arrays *and* automatic monitoring for PV variables is suppressed. The default value is 65536. To be clear: waveforms with fewer elements than this value will be automatically monitored changes, and will be converted to numpy arrays (if numpy is installed). Larger waveforms will not be automatically monitored.

Working with waveform / array data and *Strategies for working with large arrays* for more details.

4.3 Using the CA module

Many general-purpose CA functions that deal with general communication and threading contexts are very close to the C library:

`epics.ca.`**`initialize_libca`**`()`

Initialize the Channel Access library.

This loads the shared object library (DLL) to establish Channel Access Connection. The value of `PREEMPTIVE_CALLBACK` sets the pre-emptive callback model.

This must be called prior to any actual use of the CA library, but will be called automatically by the the `withCA()` decorator, so you should not need to call this directly from most real programs.

Returns **libca** : object

ca library object, used for all subsequent ca calls

See also:

`withCA` decorator to ensure CA is initialized

Notes

This function must be called prior to any real CA calls.

`epics.ca.`**`finalize_libca`**`(`*maxtime=10.0*`)`

shutdown channel access:

run `clear_channel()` for all chids in `_cache`, then calls `flush_io()` and `poll()` a few times.

Parameters **maxtime** : float

maximimum time (in seconds) to wait for `flush_io()` and `poll()` to complete.

`epics.ca.`**`context_create`**`()`

`epics.ca.`**`create_context`**`(`*ctx=None*`)`

Create a new context, using the value of `PREEMPTIVE_CALLBACK` to set the context type. Note that both *context_create* and *create_context* (which is more consistent with the Verb_Object of the rest of the CA library) are supported.

Parameters **ctx** : int

0 – No preemptive callbacks, 1 – use use preemptive callbacks, None – use value of `PREEMPTIVE_CALLBACK`

`epics.ca.`**`context_destroy`**`()`

`epics.ca.`**`destroy_context`**`()`

destroy current context

`epics.ca.`**`current_context`**`()`

return the current context

`epics.ca.`**`attach_context`**`(`*context*`)`

attach to the supplied context

`epics.ca.`**`detach_context`**`()`

detach context

`epics.ca.`**`use_initial_context`**`()`

Attaches to the context created when libca is initialized. Using this function is recommended when writing threaded programs that using CA.

See also:

Using Python Threads

`epics.ca.`**`client_status`**`(`*context, level*`)`

print (to stderr) information about Channel Access status, including status for each channel, and search and connection statistics.

`epics.ca.`**`version`**`()`
Print Channel Access version string. Currently, this should report '4.13'

`epics.ca.`**`message`**`(`*status*`)`
Print a message corresponding to a Channel Access status return value.

`epics.ca.`**`flush_io`**`()`
flush i/o

`epics.ca.`**`replace_printf_handler`**`(`*fcn=None*`)`
replace the normal printf() output handler with the supplied function (defaults to `sys.stderr.write()`)

`epics.ca.`**`pend_io`**`(`*timeout=1.0*`)`
polls CA for i/o.

`epics.ca.`**`pend_event`**`(`*timeout=1.e-5*`)`
polls CA for events

`epics.ca.`**`poll`**`(`*evt=1.e-5*[, *iot=1.0*]`)`
a convenience function which is equivalent to:: pend_event(evt) pend_io_(iot)

4.3.1 Creating and Connecting to Channels

The basic channel object is the Channel ID or `chid`. With the CA library (and `ca` module), one creates and acts on the `chid` values. These are simply `ctypes.c_long` (C long integers) that hold the memory address of the C representation of the channel, but it is probably a good idea to treat these as object instances.

`epics.ca.`**`create_channel`**`(`*pvname*, *connect=False*, *callback=None*, *auto_cb=True*`)`
create a Channel for a given pvname

creates a channel, returning the Channel ID `chid` used by other functions to identify this channel.

Parameters **pvname** : string

the name of the PV for which a channel should be created.

connect : bool

whether to (try to) connect to PV as soon as possible.

auto_cb : bool

whether to automatically use an internal connection callback.

callback : callable or `None`

user-defined Python function to be called when the connection state change s.

Returns **chid** : ctypes.c_long

channel ID.

Notes

1. The user-defined connection callback function should be prepared to accept keyword arguments of

keyword	meaning
pvname	name of PV
chid	Channel ID
conn	whether channel is connected

2. If *auto_cb* is `True`, an internal connection callback is used so that you should not need to explicitly connect to a channel, unless you are having difficulty with dropped connections.

3. If the channel is already connected for the PV name, the callback will be called immediately.

`epics.ca.`**`connect_channel`**`(`*chid*, *timeout=None*, *verbose=False*`)`

connect to a channel, waiting up to timeout for a channel to connect. It returns the connection state, `True` or `False`.

This is usually not needed, as implicit connection will be done when needed in most cases.

Parameters **chid** : ctypes.c_long

Channel ID

timeout : float

maximum time to wait for connection.

verbose : bool

whether to print out debugging information

Returns **connection_state** : bool

that is, whether the Channel is connected

Notes

1.If *timeout* is `None`, the value of *`DEFAULT_CONNECTION_TIMEOUT`* is used (defaults to 2.0 seconds).

2. Normally, channels will connect in milliseconds, and the connection callback will succeed on the first attempt.

3. For un-connected Channels (that are nevertheless queried), the 'ts' (timestamp of last connection attempt) and 'failures' (number of failed connection attempts) from the *`_cache`* will be used to prevent spending too much time waiting for a connection that may never happen.

Many other functions require a valid Channel ID, but not necessarily a connected Channel. These functions are essentially identical to the CA library versions, and include:

`epics.ca.`**`name`**`(`*chid*`)`

return PV name for channel name

`epics.ca.`**`host_name`**`(`*chid*`)`

return host name and port serving Channel

`epics.ca.`**`element_count`**`(`*chid*`)`

return number of elements in Channel's data. 1 for most Channels, > 1 for waveform Channels

`epics.ca.`**`read_access`**`(`*chid*`)`

return *read access* for a Channel: 1 for `True`, 0 for `False`.

`epics.ca.`**`write_access`**`(`*chid*`)`

return *write access* for a channel: 1 for `True`, 0 for `False`.

`epics.ca.`**`field_type`**`(`*chid*`)`

return the integer DBR field type.

See the *ftype* column from *Table of DBR Types*.

`epics.ca.`**`clear_channel`**`(`*chid*`)`

clear the channel

`epics.ca.`**`state`**`(`*chid*`)`
: return state (that is, attachment state) for channel

A few additional pythonic functions have been added:

`epics.ca.`**`isConnected`**`(`*chid*`)`
: return whether channel is connected: *dbr.CS_CONN==state(chid)*

  This is `True` for a connected channel, `False` for an unconnected channel.

`epics.ca.`**`access`**`(`*chid*`)`
: returns a string describing read/write access: one of *no access*, *read-only*, *write-only*, or *read/write*

`epics.ca.`**`promote_type`**`(`*chid*[, *use_time=False*[, *use_ctrl=False*]]`)`
: promotes the native field type of a `chid` to its TIME or CTRL variant. Returns the integer corresponding to the promoted field value.

  See *Table of DBR Types*.

`epics.ca.`**`_cache`**
: The ca module keeps a global cache of Channels that holds connection status and a bit of internal information for all known PVs. This cache is not intended for general use.

`epics.ca.`**`show_cache`**`(`*print_out=True*`)`
: print out a listing of PVs in the current session to standard output. Use the *print_out=False* option to be returned the listing instead of having it printed out.

`epics.ca.`**`clear_cache`**`()`
: Clears global caches of Epics CA connections, and fully detaches from the CA context. This is important when doing multiprocessing (and is done internally by CAProcess), but can be useful to fully reset a Channel Access session.

4.3.2 Interacting with Connected Channels

Once a `chid` is created and connected there are several ways to communicating with it. These are primarily encapsulated in the functions `get()`, `put()`, and `create_subscription()`, with a few additional functions for retrieving specific information.

These functions are where this python module differs the most from the underlying CA library, and this is mostly due to the underlying CA function requiring the user to supply DBR TYPE and count as well as `chid` and allocated space for the data. In python none of these is needed, and keyword arguments can be used to specify such options.

`epics.ca.`**`get`**`(`*chid*, *ftype=None*, *count=None*, *as_string=False*, *as_numpy=True*, *wait=True*, *timeout=None*`)`
: return the current value for a Channel. Note that there is not a separate form for array data.

  Parameters **chid** : ctypes.c_long

  Channel ID

  ftype : int

  field type to use (native type is default)

  count : int

  maximum element count to return (full data returned by default)

  as_string : bool

  whether to return the string representation of the value. See notes below.

  as_numpy : bool

whether to return the Numerical Python representation for array / waveform data.

wait : bool

whether to wait for the data to be received, or return immediately.

timeout : float

maximum time to wait for data before returning `None`.

Returns data : object

Normally, the value of the data. Will return `None` if the channel is not connected, *wait=False* was used, or the data transfer timed out.

Notes

1.Returning `None` indicates an *incomplete get*

2. The *as_string* option is not as complete as the *as_string* argument for `PV.get()`. For Enum types, the name of the Enum state will be returned. For waveforms of type CHAR, the string representation will be returned. For other waveforms (with *count* > 1), a string like *<array count=3, type=1>* will be returned.

3. The *as_numpy* option will convert waveform data to be returned as a numpy array. This is only applied if numpy can be imported.

4. The *wait* option controls whether to wait for the data to be received over the network and actually return the value, or to return immediately after asking for it to be sent. If *wait=False* (that is, immediate return), the *get* operation is said to be *incomplete*. The data will be still be received (unless the channel is disconnected) eventually but stored internally, and can be read later with `get_complete()`. Using *wait=False* can be useful in some circumstances.

5. The *timeout* option sets the maximum time to wait for the data to be received over the network before returning `None`. Such a timeout could imply that the channel is disconnected or that the data size is larger or network slower than normal. In that case, the *get* operation is said to be *incomplete*, and the data may become available later with `get_complete()`.

See *Table of DBR Types* for a listing of values of *ftype*,

See *Strategies for working with large arrays* for a discussion of strategies for how to best deal with very large arrays.

See *Strategies for connecting to a large number of PVs* for a discussion of when using *wait=False* can give a large performance boost.

See *The wait and timeout options for get(), ca.get_complete()* for further discussion of the *wait* and *timeout* options and the associated `get_complete()` function.

`epics.ca.`**`get_complete`**(*chid*, *ftype=None*, *count=None*, *as_string=False*, *as_numpy=True*, *timeout=None*)

returns the current value for a Channel, completing an earlier incomplete `get()` that returned `None`, either because *wait=False* was used or because the data transfer did not complete before the timeout passed.

Parameters chid : ctypes.c_long

Channel ID

ftype : int

field type to use (native type is default)

count : int

maximum element count to return (full data returned by default)

as_string : bool

whether to return the string representation of the value.

as_numpy : bool

whether to return the Numerical Python representation for array / waveform data.

timeout : float

maximum time to wait for data before returning `None`.

Returns data : object

This function will return `None` if the previous *`get()`* actually completed, or if this data transfer also times out.

Notes

1. The default timeout is dependent on the element count:: default_timout = 1.0 + log10(count) (in seconds)

2.Consult the doc for *`get()`* for more information.

See *The wait and timeout options for get(), ca.get_complete()* for further discussion.

`epics.ca.`**`put`**`(`*chid*, *value*, *wait=False*, *timeout=30*, *callback=None*, *callback_data=None*`)`

sets the Channel to a value, with options to either wait (block) for the processing to complete, or to execute a supplied callback function when the process has completed.

Parameters chid : ctypes.c_long

Channel ID

wait : bool

whether to wait for processing to complete (or time-out) before returning.

timeout : float

maximum time to wait for processing to complete before returning anyway.

callback : `None` of callable

user-supplied function to run when processing has completed.

callback_data : object

extra data to pass on to a user-supplied callback function.

Returns status : int

1 for success, -1 on time-out

Notes

1.Specifying a callback will override setting *wait=True*.

2.**A put-callback function will be called with keyword arguments** pvname=pvname, data=callback_data

For more on this *put callback*, see *User-supplied Callback functions* below.

`epics.ca.`**`create_subscription`**(*chid*, *use_time=False*, *use_ctrl=False*, *mask=None*, *callback=None*)

create a *subscription to changes*. Sets up a user-supplied callback function to be called on any changes to the channel.

Parameters **chid** : ctypes.c_long

channel ID

use_time : bool

whether to use the TIME variant for the PV type

use_ctrl : bool

whether to use the CTRL variant for the PV type

mask : integer or None

bitmask combination of `dbr.DBE_ALARM`, `dbr.DBE_LOG`, and `dbr.DBE_VALUE`, to control which changes result in a callback. If `None`, defaults to *`DEFAULT_SUBSCRIPTION_MASK`*.

callback : `None` or callable

user-supplied callback function to be called on changes

Returns (callback_ref, user_arg_ref, event_id)

The returned tuple contains *callback_ref* an *user_arg_ref* which are references that should be kept for as long as the subscription lives (otherwise they may be garbage collected, causing no end of trouble). *event_id* is the id for the event (useful for clearing a subscription).

Notes

Keep the returned tuple in named variable!! if the return argument gets garbage collected, a coredump will occur.

For more on writing the user-supplied callback, see *User-supplied Callback functions* below.

Warning: *event_id* is the id for the event (useful for clearing a subscription). You **must** keep the returned tuple in active variables, either as a global variable or as data in an encompassing class. If you do *not* keep this data, the return value will be garbage collected, the C-level reference to the callback will disappear, and you will see coredumps.
On Linux, a message like:

```
python: Objects/funcobject.c:451: func_dealloc: Assertion 'g->gc.gc_refs != (-2)' failed.
Abort (core dumped)
```

is a hint that you have *not* kept this data.

`epics.ca.`**`DEFAULT_SUBSCRIPTION_MASK`**

This value is the default subscription type used when calling *`create_subscription()`* with *mask=None*. It is also used by default when creating a `PV` object with auto_monitor is set to `True`.

The initial default value is *dbr.DBE_ALARM|dbr.DBE_VALUE* (i.e. update on alarm changes or value changes which exceeds the monitor deadband.) The other possible flag in the bitmask is *dbr.DBE_LOG* for archive-deadband changes.

If this value is changed, it will change the default for all subsequent calls to `create_subscription()`, but it will not change any existing subscriptions.

`epics.ca.`**`clear_subscription`**(*event_id*)
cancel subscription given its *event_id*

Several other functions are provided:

`epics.ca.`**`get_timestamp`**(*chid*)
return the timestamp of a Channel – the time of last update.

`epics.ca.`**`get_severity`**(*chid*)
return the severity of a Channel.

`epics.ca.`**`get_precision`**(*chid*)
return the precision of a Channel. For Channels with native type other than FLOAT or DOUBLE, this will be 0

`epics.ca.`**`get_enum_strings`**(*chid*)
return list of names for ENUM states of a Channel. Returns None for non-ENUM Channels

`epics.ca.`**`get_ctrlvars`**(*chid*)
return the CTRL fields for a Channel.

Depending on the native type, the keys may include *status*, *severity*, *precision*, *units*, enum_strs*, *upper_disp_limit*, *lower_disp_limit*, upper_alarm_limit*, *lower_alarm_limit*, upper_warning_limit*, *lower_warning_limit*, *upper_ctrl_limit*, *lower_ctrl_limit*

Notes

enum_strs will be a list of strings for the names of ENUM states.

See *Table of Control Attributes*

Table of Control Attributes

attribute	*data types*
status	
severity	
precision	0 for all but double, float
units	
enum_strs	enum only
upper_disp_limit	
lower_disp_limit	
upper_alarm_limit	
lower_alarm_limit	
upper_warning_limit	
lower_warning_limit	
upper_ctrl_limit	
lower_ctrl_limit	

Note that *enum_strs* will be a tuple of strings for the names of ENUM states.

`epics.ca.`**`get_timevars`**(*chid*)
returns a dictionary of TIME fields for a Channel. This will contain keys of *status*, *severity*, and *timestamp*.

4.3.3 Synchronous Groups

Synchronous Groups can be used to ensure that a set of Channel Access calls all happen together, as if in a *transaction*. Synchronous Groups work in PyEpics as of version 3.0.10, but more testing is probably needed.

The idea is to first create a synchronous group, then add a series of `sg_put()` and `sg_get()` which do not happen immediately, and finally block while all the channel access communication is done for the group as a unit. It is important to *not* issue `pend_io()` during the building of a synchronous group, as this will cause pending `sg_put()` and `sg_get()` to execute.

`epics.ca.`**`sg_create`**`()`
: create synchronous group. Returns a *group id*, *gid*, which is used to identify this group and to be passed to all other synchronous group commands.

`epics.ca.`**`sg_delete`**`(`*gid*`)`
: delete a synchronous group

`epics.ca.`**`sg_block`**`(`*gid*[, *timeout=10.0*]`)`
: block for a synchronous group to complete processing

`epics.ca.`**`sg_get`**`(`*gid*, *chid*[, *ftype=None*[, *as_string=False*[, *as_numpy=True*]]]`)`
: synchronous-group get of the current value for a Channel. same options as get()

  This function will not immediately return the value, of course, but the address of the underlying data.

  After the `sg_block()` has completed, you must use `_unpack()` to convert this data address to the actual value(s).

Examples

```
>>> chid = epics.ca.create_channel(PV_Name)
>>> epics.ca.connect_channel(chid1)
>>> sg = epics.ca.sg_create()
>>> data = epics.ca.sg_get(sg, chid)
>>> epics.ca.sg_block(sg)
>>> print epics.ca._unpack(data, chid=chid)
```

  See further example below.

`epics.ca.`**`sg_put`**`(`*gid*, *chid*, *value*`)`
: perform a *put* within a synchronous group.

  This *put* cannot wait for completion or for a a callback to complete.

`epics.ca.`**`sg_test`**`(`*gid*`)`
: test whether a synchronous group has completed.

`epics.ca.`**`sg_reset`**`(`*gid*`)`
: resets a synchronous group

An example use of a synchronous group:

```
from epics import ca
import time

pvs = ('X1.VAL', 'X2.VAL', 'X3.VAL')
chids = [ca.create_channel(pvname) for pvname in pvs]

for chid in chids:
    ca.connect_channel(chid)
    ca.put(chid, 0)

# create synchronous group
sg = ca.sg_create()

# get data pointers from ca.sg_get
data = [ca.sg_get(sg, chid) for chid in chids]

print 'Now change these PVs for the next 10 seconds'
time.sleep(10.0)

print 'will now block for i/o'
ca.sg_block(sg)
#
# CALL ca._unpack with data points and chid to extract data
for pvname, dat, chid in zip(pvs, data, chids):
    val = ca._unpack(dat, chid=chid)
    print "%s = %s" % (pvname, str(val))

ca.sg_reset(sg)

#  Now a SG Put
print 'OK, now we will put everything back to 0 synchronously'

for chid in chids:
    ca.sg_put(sg, chid, 0)

print 'sg_put done, but not blocked / committed. Sleep for 5 seconds '
time.sleep(5.0)
ca.sg_block(sg)
print 'done.'
```

4.4 Implementation details

The details given here should mostly be of interest to those looking at the implementation of the *ca* module, those interested in the internals, or those looking to translate lower-level C or Python code to this module.

4.4.1 DBR data types

Table of DBR Types

CA type	*integer ftype*	*Python ctypes type*
string	0	string
int	1	integer
short	1	integer
float	2	double
enum	3	integer
char	4	byte
long	5	integer
double	6	double
time_string	14	
time_int	15	
time_short	15	
time_float	16	
time_enum	17	
time_char	18	
time_long	19	
time_double	20	
ctrl_string	28	
ctrl_int	29	
ctrl_short	29	
ctrl_float	30	
ctrl_enum	31	
ctrl_char	32	
ctrl_long	33	
ctrl_double	34	

4.4.2 *PySEVCHK* and ChannelAccessExcepction: checking CA return codes

exception `epics.ca.`**`ChannelAccessException`**

This exception is raised when the `ca` module experiences unexpected behavior and must raise an exception

`epics.ca.`**`PySEVCHK`**(*func_name*, *status*[, *expected=dbr.ECA_NORMAL*])

This checks the return *status* returned from a *libca.ca_**** and raises a `ChannelAccessException` if the value does not match the *expected* value (which is nornmally `dbr.ECA_NORMAL`.

The message from the exception will include the *func_name* (name of the Python function) and the CA message from `message()`.

`epics.ca.`**`withSEVCHK`**(*fcn*)

decorator to raise a ChannelAccessException if the wrapped ca function does not return status = dbr.ECA_NORMAL. This handles the common case of running `PySEVCHK()` for a function whose return value is from a corresponding libca function and whose return value should be `dbr.ECA_NORMAL`.

4.4.3 Function Decorators

In addition to `withSEVCHK()`, several other decorator functions are used heavily inside of ca.py or are available for your convenience.

`epics.ca.`**`withCA`**(*fcn*)

decorator to ensure that libca and a context are created prior to function calls to the channel access library. This is intended for functions that need CA started to work, such as `create_channel()`.

Note that CA functions that take a Channel ID (chid) as an argument are NOT wrapped by this: to get a chid, the library must have been initialized already.

`epics.ca.`**`withCHID`**(*fcn*)

decorator to ensure that first argument to a function is a Channel ID, `chid`. The test performed is very weak, as any ctypes long or python int will pass, but it is useful enough to catch most accidental errors before they would cause a crash of the CA library.

`epics.ca.`**`withConnectedCHID`**(*fcn*)

decorator to ensure that the first argument of a function is a fully connected Channel ID, `chid`. This test is (intended to be) robust, and will try to make sure a `chid` is actually connected before calling the decorated function.

`epics.ca.`**`withInitialContext`**(*fcn*)

decorator to ensure that the wrapped function uses the initial threading context created at initialization of CA

See *Using Python Threads* for further discussion.

4.4.4 Unpacking Data from Callbacks

Throughout the implementation, there are several places where data returned by the underlying CA library needs to be be converted to Python data. This is encapsulated in the `_unpack()` function. In general, you will not have to run this code, but there is one exception: when using `sg_get()`, the values returned will have to be unpacked with this function.

`epics.ca.`**`_unpack`**(*chid*, *data*[, *count=None*[, *ftype=None*[, *as_numpy=None*]]])

unpacks raw data for a Channel ID *chid* returned by libca functions including *ca_get_array_callback* or subscription callback, and returns the corresponding Python data

Normally, users are not expected to need to access this function, but it will be necessary why using `sg_get()`.

Parameters **chid** : ctypes.c_long or `None`

channel ID (if not None, used for determining count and ftype)

data : object

raw data as returned by internal libca functions.

count : integer

number of elements to fetch (defaults to element count of chid or 1)

ftype : integer

data type of channel (defaults to native type of chid)

as_numpy : bool

whether to convert to numpy array.

4.5 User-supplied Callback functions

User-supplied callback functions can be provided for both `put()` and `create_subscription()`. Note that callbacks for *PV* objects are slightly different: see *User-supplied Callback functions* in the `pv` module for details.

When defining a callback function to be run either when a `put()` completes or on changes to the Channel, as set from `create_subscription()`, it is important to know two things:

1. how your function will be called.
2. what is permissible to do inside your callback function.

In both cases, callbacks will be called with keyword arguments. You should be prepared to have them passed to your function. Use ***kw* unless you are very sure of what will be sent.

For callbacks sent when a `put()` completes, your function will be passed these:

- *pvname* : the name of the pv
- *data*: the user-supplied callback_data (defaulting to `None`).

For subscription callbacks, your function will be called with keyword/value pairs that will include:

- *pvname*: the name of the pv
- *value*: the latest value
- *count*: the number of data elements
- *ftype*: the numerical CA type indicating the data type
- *status*: the status of the PV (1 for OK)
- *chid*: the integer address for the channel ID.

Depending on the data type, and whether the CTRL or TIME variant was used, the callback function may also include some of these as keyword arguments:

- *enum_strs*: the list of enumeration strings
- *precision*: number of decimal places of precision.
- *units*: string for PV units
- *severity*: PV severity
- *timestamp*: timestamp from CA server.

Note that a the user-supplied callback will be run *inside* a CA function, and cannot reliably make any other CA calls. It is helpful to think "this all happens inside of a `pend_event()` call", and in an epics thread that may or may not be the main thread of your program. It is advisable to keep the callback functions short and not resource-intensive. Consider strategies which use the callback only to record that a change has occurred and then act on that change later – perhaps in a separate thread, perhaps after `pend_event()` has completed.

4.6 Omissions

Several parts of the CA library are not implemented in the Python module. These are currently seen as unneeded (with notes where appropriate for alternatives), though they could be added on request.

`epics.ca.`**`ca_add_exception_event`**`()`
Not implemented: Python exceptions are raised where appropriate and can be used in user code.

`epics.ca.`**`ca_add_fd_registration`**`()`
Not implemented

`epics.ca.`**`ca_replace_access_rights_event`**`()`
Not implemented

`epics.ca.`**`ca_client_status`**`()`
Not implemented

`epics.ca.`**`ca_set_puser`**`()`
Not implemented : it is easy to pass user-defined data to callbacks as needed.

`epics.ca.`**`ca_puser`**`()`
Not implemented: it is easy to pass user-defined data to callbacks as needed.

epics.ca.**ca_SEVCHK**()
Not implemented: the Python function `PySEVCHK()` is approximately the same.

epics.ca.**ca_signal**()
Not implemented: the Python function `PySEVCHK()` is approximately the same.

epics.ca.**ca_test_event**()
Not implemented: this appears to be a function for debugging events. These are easy enough to simulate by directly calling Python callback functions.

epics.ca.**ca_dump_dbr**()
Not implemented

In addition, not all *DBR* types in the CA C library are supported.

Only native types and their DBR_TIME and DBR_CTRL variants are supported: DBR_STS and DBR_GR variants are not. Several *dbr_XXX* functions are also not supported, as they are needed only to dynamically allocate memory.

4.7 `CAThread` class

class epics.ca.**CAThread**(*group=None*[, *target=None*[, *name=None*[, *args=()*[, *kwargs={}*]]]])
create a CA-aware subclass of a standard Python `threading.Thread`. See the standard library documentation for further information on how to use Thread objects.

A *CAThread* simply runs `use_initial_context()` prior to running each target function, so that `use_initial_context()` does not have to be explicitly put inside the target function.

The See *Using Python Threads* for further discussion.

4.8 Examples

Here are some example sessions using the `ca` module.

4.8.1 Create, Connect, Get Value of Channel

Note here that several things have been simplified compare to using CA in C: initialization and creating a main-thread context are handled, and connection of channels is handled in the background:

```
from epics import ca
chid  = ca.create_channel('XXX:m1.VAL')
count = ca.element_count(chid)
ftype = ca.field_type(chid)
print "Channel ", chid, count, ftype
value = ca.get()
print value
```

4.8.2 Put, waiting for completion

Here we set a PVs value, waiting for it to complete:

```
from epics import ca
chid  = ca.create_channel('XXX:m1.VAL')
ca.put(chid,  1.0, wait=True)
```

The `put()` method will wait to return until the processing is complete.

4.8.3 Define a callback to Subscribe to Changes

Here, we *subscribe to changes* for a PV, which is to say we define a callback function to be called whenever the PV value changes. In the case below, the function to be called will simply write the latest value out to standard output:

```
from epics import ca
import time
import sys

# define a callback function.  Note that this should
# expect certain keyword arguments, including 'pvname' and 'value'
def onChanges(pvname=None, value=None, **kw):
    fmt = 'New Value: %s  value=%s, kw=%s\n'
    sys.stdout.write(fmt % (pvname, str(value), repr(kw)))
    sys.stdout.flush()

# create the channel
mypv = 'XXX.VAL'
chid = ca.create_channel(mypv)

# subscribe to events giving a callback function
eventID = ca.create_subscription(chid, callback=onChanges)

# now we simply wait for changes
t0 = time.time()
while time.time()-t0 < 10.0:
    time.sleep(0.001)
```

It is **vital** that the return value from `create_subscription()` is kept in a variable so that it cannot be garbage collected. Failure to keep this value will cause trouble, including almost immediate segmentation faults (on Windows) or seemingly inexplicable crashes later (on linux).

4.8.4 Define a connection callback

Here, we define a connection callback – a function to be called when the connection status of the PV changes. Note that this will be called on initial connection:

```
import epics
import time

def onConnectionChange(pvname=None, conn=None, chid=None):
    print 'ca connection status changed:  ', pvname,  conn, chid

# create channel, provide connection callback
motor1 = '13IDC:m1'
chid = epics.ca.create_channel(motor1, callback=onConnectionChange)

print 'Now waiting, watching values and connection changes:'
t0 = time.time()
while time.time()-t0 < 30:
    time.sleep(0.001)
```

This will run the supplied callback soon after the channel has been created, when a successful connection has been made. Note that the callback should be prepared to accept keyword arguments of *pvname*, *chid*, and *conn* for the PV name, channel ID, and connection state (`True` or `False`).

CHAPTER

FIVE

WORKING WITH WAVEFORM / ARRAY DATA

Though most EPICS Process Variables hold single values, PVs can hold array data from EPICS waveform records. These are always data of a homogenous data type, and have a fixed maximum element count (defined when the waveform is created from the host EPICS process). Epics waveforms are most naturally mapped to Arrays from the numpy module, and this is strongly encouraged.

5.1 Arrays without Numpy

If you have numpy installed, and use the default *as_numpy=True* in `ca.get()`, `pv.get()` or `epics.caget()`, you will get a numpy array for the value of a waveform PV. If you do *not* have numpy installed, or explicitly use *as_numpy=False* in a get request, you will get the raw C-like array reference from the Python ctypes module. These objects are not normally meant for casual use, but are not too difficult to work with either. They can be easily converted to a simple Python list with something like:

```
>>> import epics
>>> epics.ca.HAS_NUMPY = False # turn numpy off for session
>>> p = epics.PV('XX:scan1.P1PA')
>>> p.get()
<epics.dbr.c_double_Array_500 object at 0x853980c>
>>> ldat = list(p.get())
```

Note that this conversion to a list can be very slow for large arrays.

5.2 Variable Length Arrays: NORD and NELM

While the maximum length of an array is fixed, the length of data you get back from a monitor, `ca.get()`, `pv.get()`, or `epics.caget()` may be shorter than the maximumn length, reflecting the most recent data put to that PV. That is, if some process puts a smaller array to a PV than its maximum length, monitors on that PV may receive only the changed data. For example:

```
>>> import epics
>>> p = epics.PV('Py:double2k')
>>> print p
<PV 'Py:double2k', count=2048/2048, type=double, access=read/write>
>>> import numpy
>>> p.put(numpy.arange(10)/5.0)
>>> print p.get()
array([ 0. ,  0.2,  0.4,  0.6,  0.8,  1. ,  1.2,  1.4,  1.6,  1.8])
```

To be clear, the `pv.put()` above could be done in a separate process – the `pv.get()` is not using a value cached from the `pv.put()`.

This feature seems to depend on the record definition, and requires version 3.14.12.1 of Epics base or higher, and can be checked by comparing `ca.version()` with the string '4.13'.

5.3 Character Arrays

As noted in other sections, character waveforms can be used to hold strings longer than 40 characters, which is otherwise a fundamental limit for native Epics strings. Character waveforms shorter than `ca.AUTOMONITOR_MAXLENGTH` can be turned into strings with an optional *as_string=True* to `ca.get()`, `pv.get()` , or `epics.caget()`. If you've defined a Epics waveform record as:

```
record(waveform,"$(P):filename")  {
          field(DTYP,"Soft Channel")
          field(DESC,"file name")
          field(NELM,"128")
          field(FTVL,"CHAR")
 }
```

Then you can use this record with:

```
>>> import epics
>>> pvname = 'PREFIX:filename.VAL'
>>> pv   = epics.PV(pvname)
>>> print pv.info
....
>>> plain_val = pv.get()
>>> print plain_val
array([ 84,  58,  92, 120,  97, 115,  95, 117, 115, 101, 114,  92,  77,
     97, 114,  99, 104,  50,  48,  49,  48,  92,  70,  97, 115, 116,
     77,  97, 112])
>>> char_val = pv.get(as_string=True)
>>> print char_val
'T:\\xas_user\\March2010\\FastMap'
```

This example uses `pv.get()` but `ca.get()` is essentially equivalent, as its *as_string* parameter works exactly the same way.

Note that Epics character waveforms as defined as above are really arrays of bytes. The converion to a string assumes the ASCII character set. Unicode is not directly supported. If you are storing non-ASCII data, you would have to convert the raw array data yourself, perhaps like this (for Python3):

```
>>> arr_data = pv.get()
>>> arr_bytes = bytes(list(array_data))
>>> arr_string = str(arr_bytes, 'LATIN-1')
```

5.3.1 Strategies for working with large arrays

EPICS Channels / Process Variables usually have values that can be stored with a small number of bytes. This means that their storage and transfer speeds over real networks is not a significant concern. However, some Process Variables can store much larger amounts of data (say, several megabytes) which means that some of the assumptions about dealing with Channels / PVs may need reconsideration.

When using PVs with large array sizes (here, I'll assert that *large* means more than a few thousand elements), it is necessary to make sure that the environmental variable `EPICS_CA_MAX_ARRAY_BYTES` is suitably set. Unfortunately, this represents a pretty crude approach to memory management within Epics for handling array data as it is used not only sets how large an array the client can accept, but how much memory will be allocated on the server. In addition, this value must be set prior to using the CA library – it cannot be altered during the running of a CA program.

Normally, the default value for `EPICS_CA_MAX_ARRAY_BYTES` is 16384 (16k, and it turns out that you cannot set it smaller than this value!). As Python is used for clients, generally running on workstations or servers with sufficient memory, this default value is changed to 2**24, or 16Mb) when *epics.ca* is initialized. If the environmental variable `EPICS_CA_MAX_ARRAY_BYTES` has not already been set.

The other main issue for PVs holding large arrays is whether they should be automatically monitored. For PVs holding scalar data or small arrays, any penalty for automatically monitoring these variables (that is, causing network traffic every time a PV changes) is a small price to pay for being assured that the latest value is always available. As arrays get larger (as for data streams from Area Detectors), it is less obvious that automatic monitoring is desirable.

The Python *epics.ca* module defines a variable `ca.AUTOMONITOR_MAXLENGTH` which controls whether array PVs are automatically monitored. The default value for this variable is 65536, but can be changed at runtime. Arrays with fewer elements than `ca.AUTOMONITOR_MAXLENGTH` will be automatically monitored, unless explicitly set, and arrays larger than `AUTOMONITOR_MAXLENGTH` will not be automatically monitored unless explicitly set. Auto-monitoring of PVs can be be explicitly set with

```
>>> pv2 = epics.PV('ScalerPV', auto_monitor=True)
>>> pv1 = epics.PV('LargeArrayPV', auto_monitor=False)
```

5.4 Example handling Large Arrays

Here is an example reading data from an EPICS areaDetector, as if it were an image from a digital camera. This uses the Python Imaging Library for much of the image processing:

```
>>> import epics
>>> import Image
>>> pvname = '13IDCPS1:image1:ArrayData'
>>> img_pv  = epics.PV(pvname)
>>>
>>> raw_image = img_pv.get()
>>> im_mode = 'RGB'
>>> im_size = (1360, 1024)
>>> img = Image.frombuffer(im_mode, im_size, raw_image,
                           'raw', im_mode, 0, 1)
>>> img.show()
```

The result looks like this (taken with a Prosilica GigE camera):

A more complete application for reading and displaying image from Epics Area Detectors is included at http://github.com/pyepics/epicsapps/.

CHAPTER

SIX

DEVICES: COLLECTIONS OF PVS

6.1 Overview

The `device` module provides a simple interface to a collection of PVs. Here an epics `device.Device` is an object holding a set of PVs, all sharing a prefix, but having many *attributes*. Many PVs will have names made up of *prefix+attribute*, with a common prefix for several related PVs. This almost describes an Epics Record, but as it is concerned only with PV names, the mapping to an Epics Record is not exact. On the other hand, the concept of a *device* is more flexible than a predefined Epics Record as it can actually hold PVs from several different records.:

```
motor1 = epics.Device('XXX:motor1.', attrs=('VAL', 'RBV', 'DESC', 'RVAL',
                                            'LVIO', 'HLS', 'LLS'))
motor1.put('VAL', 1)
print 'Motor %s = %f' % ( motor1.get('DESC'), motor1.get('RBV'))

motor1.VAL = 0
print 'Motor %s = %f' % ( motor1.DESC, motor1.RBV )
```

While useful on its own like this, the real point of a *device* is as a base class, to be inherited and extended. In fact, there is a more sophisticated Motor device described below at *Epics Motor Device*

class `device.`**`Device`**(*prefix=None*[, *delim=''*[, *attrs=None*]])

The attribute PVs are built as needed and held in an internal buffer `self._pvs`. This class is kept intentionally simple so that it may be subclassed.

To pre-load attribute names on initialization, provide a list or tuple of attributes with the *attr* option.

Note that *prefix* is actually optional. When left off, this class can be used as an arbitrary container of PVs, or to turn any subclass into an epics Device.

In general, PV names will be mapped as prefix+delim+attr. See `add_pv()` for details of how to override this.

`device.`**`PV`**(*attr[, connect=True[, **kw]]]*)

returns the *PV* object for a device attribute. The connect argument and any other keyword arguments are passed to `epics.PV()`.

`device.`**`put`**(*attr*, *value*[, *wait=False*[, *timeout=10.0*]])

put an attribute value, optionally wait for completion or up to a supplied timeout value

`device.`**`get`**(*attr*[, *as_string=False*])

get an attribute value, option as_string returns a string representation

`device.`**`add_callback`**(*attr*, *callback*)

add a callback function to an attribute PV, so that the callback function will be run when the at tribute's value changes

device.**add_pv**(*pvname*[, *attr=None*[, ***kw*]])
adds an explicitly names `epics.PV()` to the device even though it may violate the normal naming rules (in which *attr* is mapped to *epics.PV(prefix+delim+attr)*. That is, one can say:

```
import epics
m1 = epics.Device('XXX:m1', delim='.')
m1.add_pv('XXX:m2.VAL', attr='other')
print m1.VAL      # print value of XXX:m1.VAL
print m1.other    # prints value of XXX:m2.VAL
```

device.**save_state**()
return a dictionary of all current values – the ''current state''.

device.**restore_state**(*state*)
restores a saved state, as saved with *save_state()*

device.**write_state**(*fname*[, *state=None*])
write a saved state to a file. If no state is provide, the current state is written.

device.**read_state**(*fname*[, *restore=False*])
reads a state from a file, as written with *write_state()*, and returns it. If ''restore'' is `True`, the read state will be restored.

device.**_pvs**
a dictionary of PVs making up the device.

6.2 Epics Motor Device

The Epics Motor record has over 100 fields associated with it. Of course, it is often preferable to think of 1 Motor with many attributes than 100 or so separate PVs. Many of the fields of the Motor record are interrelated and influence other settings, including limits on the range of motion which need to be respected, and which may send notifications when they are violated. Thus, there is a fair amount of functionality for a Motor. Typically, the user just wants to move the motor by setting its drive position, but a fully enabled Motor should allow the use to change and read many of the Motor parameters.

The *Motor* class helps the user create and use Epics motors. A simple example use would be:

```
import epics
m1 = epics.Motor('XXX:m1')

print 'Motor:  ', m1.DESC , ' Currently at ', m1.RBV

m1.tweak_val = 0.10
m1.move(0.0, dial=True, wait=True)

for i in range(10):
    m1.tweak(direction='forward', wait=True)
    time.sleep(1.0)
    print 'Motor:  ', m1.DESC , ' Currently at ', m1.RBV
```

Which will step the motor through a set of positions. You'll notice a few features for Motor:

1. Motors can use English-name aliases for attributes for fields of the motor record. Thus 'VAL' can be spelled 'drive' and 'DESC' can be 'description'. The *Table of Motor Attributes* give the list of names that can be used.

2. The methods for setting positions can use the User, Dial, or Step coordinate system, and can wait for completion.

6.2.1 The `epics.Motor` class

class `motor.`**`Motor`** (*pvname*[, *timeout=30.*])
create a Motor object for a named Epics Process Variable.

Parameters

- **`pvname`** (*string*) – prefix name (no '.VAL' needed!) of Epics Process Variable for a Motor
- **`timeout`** (*float*) – time (in seconds) to wait before giving up trying to connect.

Once created, a Motor should be ready to use.

```
>>> from epics import Motor
>>> m = Motor('XX:m1')
>>> print m.drive, m.description, m.slew_speed
1.030 Fine X 5.0
>>> print m.get('device_type', as_string=True)
'asynMotor'
```

A Motor has very many fields. Only a few of them are created on initialization – the rest are retrieved as needed. The motor fields can be retrieved either with an attribute or with the `get()` method. A full list of Motor attributes and their aliases for the motor record is given in *Table of Motor Attributes*.

Table of Aliases for attributes for the epics `Motor` class, and the corresponding attribute name of the Motor Record field.

alias	*Motor Record field*		**alias**	*Motor Record field*
disabled	_able.VAL		moving	MOVN
acceleration	ACCL		resolution	MRES
back_accel	BACC		motor_status	MSTA
backlash	BDST		offset	OFF
back_speed	BVEL		output_mode	OMSL
card	CARD		output	OUT
dial_high_limit	DHLM		prop_gain	PCOF
direction	DIR		precision	PREC
dial_low_limit	DLLM		readback	RBV
settle_time	DLY		retry_max	RTRY
done_moving	DMOV		retry_count	RCNT
dial_readback	DRBV		retry_deadband	RDBD
description	DESC		dial_difference	RDIF
dial_drive	DVAL		raw_encoder_pos	REP
units	EGU		raw_high_limit	RHLS
encoder_step	ERES		raw_low_limit	RLLS
freeze_offset	FOFF		relative_value	RLV
move_fraction	FRAC		raw_motor_pos	RMP
hi_severity	HHSV		raw_readback	RRBV
hi_alarm	HIGH		readback_res	RRES
hihi_alarm	HIHI		raw_drive	RVAL
high_limit	HLM		dial_speed	RVEL
high_limit_set	HLS		s_speed	S
hw_limit	HLSV		s_back_speed	SBAK
home_forward	HOMF		s_base_speed	SBAS
home_reverse	HOMR		s_max_speed	SMAX
high_op_range	HOPR		set	SET

Continued on next page

Table 6.1 – continued from previous page

alias	*Motor Record field*		alias	*Motor Record field*
high_severity	HSV		stop_go	SPMG
integral_gain	ICOF		s_revolutions	SREV
jog_accel	JAR		stop	STOP
jog_forward	JOGF		t_direction	TDIR
jog_reverse	JOGR		tweak_forward	TWF
jog_speed	JVEL		tweak_reverse	TWR
last_dial_val	LDVL		tweak_val	TWV
low_limit	LLM		use_encoder	UEIP
low_limit_set	LLS		u_revolutions	UREV
lo_severity	LLSV		use_rdbl	URIP
lolo_alarm	LOLO		drive	VAL
low_op_range	LOPR		base_speed	VBAS
low_alarm	LOW		slew_speed	VELO
last_rel_val	LRLV		version	VERS
last_dial_drive	LRVL		max_speed	VMAX
last_SPMG	LSPG		use_home	ATHM
low_severity	LSV		deriv_gain	DCOF

6.2.2 methods for `epics.Motor`

`motor.`**`get`**`(`*attr*[, *as_string=False*]`)`

sets a field attribute for the motor.

Parameters

- **`attr`** (*string (from table above)*) – attribute name
- **`as_string`** (`True`/ `False`) – whether to return string value.

Note that `get()` can return the string value, while fetching the attribute cannot do so:

```
>>> m = epics.Motor('XXX:m1')
>>> print m.device_type
0
>>> print m.get('device_type', as_string=True)
'asynMotor'
```

`motor.`**`put`**`(`*attr*, *value*[, *wait=False*[, *timeout=30*]]`)`

sets a field attribute for the motor.

Parameters

- **`attr`** (*string (from table above)*) – attribute name
- **`value`** – value for attribute
- **`wait`** (`True`/`False`) – whether to wait for completion.
- **`timeout`** (*float*) – time (in seconds) to wait before giving up trying to connect.

`motor.`**`check_limits`**`()`

checks whether the current motor position is causing a motor limit violation, and raises a MotorLimitException if it is.

returns `None` if there is no limit violation.

`motor.`**`within_limits`**(*value*[, *limits='user'*])
checks whether a target value **would be** a limit violation.

Parameters

- **`value`** – target value
- **`limits`** (*string*) – one of 'user', 'dial', or 'raw' for which limits to consider

Return type `True/False`

`motor.`**`move`**(*val=None*[, *relative=None*[, *wait=False*[, *timeout=300.0*[, *dial=False*[, *raw=False*[, *ignore_limits=False*[, *confirm_move=False*]]]]]]])
moves motor to specified drive position.

Parameters

- **`val`** – value to move to (float) [Must be provided]
- **`relative`** – move relative to current position (T/F) [F]
- **`wait`** – whether to wait for move to complete (T/F) [F]
- **`timeout`** – max time for move to complete (in seconds) [300]
- **`dial`** – use dial coordinates (T/F) [F]
- **`raw`** – use raw coordinates (T/F) [F]
- **`ignore_limits`** – try move without regard to limits (T/F) [F]
- **`confirm_move`** – try to confirm that move has begun (when wait=False) (T/F) [F]

Return type integer

Returns an integer value, according the table below. Note that a return value of 0 with *wait=False* does not really guarantee a successful move, just that a move request was issued. If you're interested in checking that a requested move really did start without waiting for the move to complete, you may want to use the *confirm_move=True* option.

Table of return values from `move()`.

return value	meaning
-13	invalid value (cannot convert to float). Move not attempted.
-12	target value outside soft limits. Move not attempted.
-11	drive PV is not connected: Move not attempted.
-8	move started, but timed-out.
-7	move started, timed-out, but appears done.
-5	move started, unexpected return value from `put()`
-4	move-with-wait finished, soft limit violation seen.
-3	move-with-wait finished, hard limit violation seen.
0	move-with-wait finish OK.
0	move-without-wait executed, not cpmfirmed.
1	move-without-wait executed, move confirmed.
3	move-without-wait finished, hard limit violation seen.
4	move-without-wait finished, soft limit violation seen.

`motor.`**`tweak`**(*direction='forward'*[, *wait=False*[, *timeout=300.*]])
move the motor by the current *tweak value*

Parameters

- **`direction`** (*string: 'forward' (default) or 'reverse'*) – direction of motion

- **wait** (True/False) – whether to wait for completion
- **timeout** (*float*) – max time for move to complete (in seconds) [default=300]

motor.**get_position**(*readback=False*[, *dial=False*[, *raw=False*]])
Returns the motor position in user, dial or raw coordinates.

Parameters

- **readback** – whether to return the readback position in the desired coordinate system. The default is to return the drive position of the motor.
- **dial** – whether to return the position in dial coordinates. The default is user coordinates.
- **raw** – whether to return the raw position. The default is user coordinates.

The "raw" and "dial" keywords are mutually exclusive. The "readback" keyword can be used in user, dial or raw coordinates.

motor.**set_position**(*position[dial=False[, raw=False]]*)
set (that is, redefine) the current position to supplied value.

Parameters

- **position** – The new motor position
- **dial** – whether to set in dial coordinates. The default is user coordinates.
- **raw** – whether to set in raw coordinates. The default is user coordinates.

  The 'raw' and 'dial' keywords are mutually exclusive.

motor.**get_pv**(*attr*)
returns the *PV* for the corresponding attribute.

motor.**set_callback**(*attr='drive'*[, *callback=None*[, *kw=None*]])
sets a callback on the *PV* for a particular attribute.

motor.**clear_callback**(*attr='drive'*)
clears a callback on the *PV* for a particular attribute.

motor.**show_info**()
prints out a table of attributes and their current values.

6.3 Other Device Examples

An epics Device provides a general way to group together a set of PVs. The examples below show how to build on this generality, and may inspire you to build your own device classes.

6.3.1 A basic Device without a prefix

Here, we define a very simple device that does not even define a prefix. This is not much more than a collection of PVs. Since there is no prefix given, all PVs in the device must be *fully qualified*. Note that there is no requirement to share a common prefix in such a collection of PVs:

```
from epics import Device
dev = Device()
p1 = dev.PV('13IDC:m1.VAL')
p2 = dev.PV('13IDC:m2.VAL')
dev.put('13IDC:m1.VAL', 2.8)
dev.put('13IDC:m2.VAL', 3.0)
print dev.PV('13IDC:m3.DIR').get(as_string=True)
```

Note that this device cannot use the attributes based on field names.

This may not look very interesting – why not just use a bunch of PVs? If ou consider *Device* to be a starting point for building more complicated objects by subclassing *Device* and adding specialized methods, then it can start to get interesting.

6.3.2 Epics ai record as Device

For a slightly more useful and typical example, the pyepics distribution includes a Device for an Epics ai (analog input record). The full implementation of this device is:

```
#!/usr/bin/env python
"""Epics analog input record"""
from .. import Device

class ai(Device):
    "Simple analog input device"

    attrs = ('VAL', 'EGU', 'HOPR', 'LOPR', 'PREC', 'NAME', 'DESC',
             'DTYP', 'INP', 'LINR', 'RVAL', 'ROFF', 'EGUF', 'EGUL',
             'AOFF', 'ASLO', 'ESLO', 'EOFF', 'SMOO', 'HIHI', 'LOLO',
             'HIGH', 'LOW', 'HHSV', 'LLSV', 'HSV', 'LSV', 'HYST')

    def __init__(self, prefix, **kwargs):
        if prefix.endswith('.'):
            prefix = prefix[:-1]
        Device.__init__(self, prefix, delim='.', attrs=self.attrs, **kwargs)
```

The code simply pre-defines the fields that are the *suffixes* of an Epics ai input record, and subclasses `Device` with these fields to create the corresponding PVs. For most record suffixes, these will be available as attributes of the Device object. For example, the `ai` class above can be used simply and cleanly as:

```
from epics.devices import ai
This_ai = ai('XXX.PRES')
print 'Value: ', This_ai.VAL
print 'Units: ', This_ai.EGU
```

Of course, you can also use the *get ()*, *put ()* methods above for a basic `Device`:

```
This_ai.put('DESC', 'My Pump')
```

Several of the other standard Epics records can easily be exposed as Devices in this way, and the pyepics distribution includes such simple wrappings for the Epics ao, bi, and bo records, as well as several more complex records from synApps.

6.3.3 Epics Scaler Record as Device

For a slightly more complicated example: an incomplete, but very useful mapping of the Scaler Record from synApps, including methods for changing modes, and reading and writing data.

```
#!/usr/bin/env python
"""Epics Scaler"""
from .. import Device, poll
```

```
class Scaler(Device):
    """
    Simple implementation of SynApps Scaler Record.
    """
    attrs = ('CNT', 'CONT', 'TP', 'T', 'VAL')
    attr_kws = {'calc_enable': '%s_calcEnable.VAL'}
    chan_attrs = ('NM%i', 'S%i')
    calc_attrs = {'calc%i': '%s_calc%i.VAL', 'expr%i': '%s_calc%i.CALC'}
    _nonpvs = ('_prefix', '_pvs', '_delim', '_nchan', '_chans')

    def __init__(self, prefix, nchan=8):
        self._nchan  = nchan
        self._chans = range(1, nchan+1)

        attrs = list(self.attrs)
        for i in self._chans:
            for att in self.chan_attrs:
                attrs.append(att % i)

        Device.__init__(self, prefix, delim='.', attrs=attrs)

        for key, val in self.attr_kws.items():
            self.add_pv(val % prefix, attr= key)

        for i in self._chans:
            for key, val in self.calc_attrs.items():
                self.add_pv(val % (prefix, i), attr = key % i)
        self._mutable = False

    def AutoCountMode(self):
        "set to autocount mode"
        self.put('CONT', 1)

    def OneShotMode(self):
        "set to one shot mode"
        self.put('CONT', 0)

    def CountTime(self, ctime):
        "set count time"
        self.put('TP', ctime)

    def Count(self, ctime=None, wait=False):
        "set count, with optional counttime"
        if ctime is not None:
            self.CountTime(ctime)
        self.put('CNT', 1, wait=wait)
        poll()

    def EnableCalcs(self):
        " enable calculations"
        self.put('calc_enable', 1)

    def setCalc(self, i, calc):
        "set the calculation for scaler i"
        attr = 'expr%i'  % i
        self.put(attr, calc)

    def getNames(self):
        "get all names"
        return [self.get('NM%i' % i) for i in self._chans]

    def Read(self, use_calc=False):
        "read all values"
        attr = 'S%i'
        if use_calc:
            attr = 'calc%i'
        return [self.get(attr % i) for i in self._chans]
```

Note that we can then create a `scaler` object from its base PV prefix, and use methods like `Count()` and `Read()` without directly invoking epics calls:

```
s1 = Scaler('XXX:scaler1')
s1.setCalc(2, '(B-2000*A/10000000.)')
s1.enableCalcs()
s1.OneShotMode()
s1.Count(t=5.0, wait=True)
print 'Names:       ', s1.getNames()
print 'Raw  values: ', s1.Read(use_calc=False)
print 'Calc values: ', s1.Read(use_calc=True)
```

6.3.4 Other Devices included in PyEpics

Several other Epics Records have been exposed as Devices, and included in PyEpics distribution. These vary some in how complete and feature-rich they are, and are definitely skewed toward data collection at synchrotron beamlines. A table of current Devices are listed in the *Table of Included Epics Devices* table below. For further details, consult the source code for these modules.

Table of Epics Devices Included in the PyEpics distribtion. For those described as "pretty basic", there are generally only PV suffixes to attributes mapped. Many of the others include one or more methods for specific use of that Device.

module	**class**	description
ad_base	AD_Camera	areaDetector Camera, pretty basic
ad_fileplugin	AD_FilePlugin	areaDetector File Plugin, many methods
ad_image	AD_ImagePlugin	areaDetector Image, with ArrayData attribute
ad_overlay	AD_OverlayPlugin	areaDetector Overlay, pretty basic
ad_perkinelmer	AD_PerkinElmer	PerkinElmer(xrd1600) detector, several methods
ai	ai	analog input, pretty basic (as above)
ao	ao	analog output, pretty basic
bi	bi	binary input, pretty basic
bo	bo	binary output, pretty basic
mca	MCA	epics DXP record, pretty basic
mca	DXP	epics MCA record, get_rois()/get_calib()
mca	MultiXMAP	Multiple XIA XMaps, several methods
scaler	Scaler	epics Scaler record, many methods
scan	Scan	epics SScan record, some methods
srs570	SRS570	SRS570 Amplifier
struck	Struck	SIS Multichannel Scaler, many methods
transform	Transform	epics userTransform record

CHAPTER

SEVEN

ALARMS: RESPOND WHEN A PV GOES OUT OF RANGE

7.1 Overview

The *alarm* module provides an Alarm object to specify an alarm condition and what to do when that condition is met.

class alarm.**Alarm**(*pvname*[, *comparison=None*[, *trip_point=None*[, *callback=None*[, *alert_delay=10*]]]])

creates an alarm object.

param pvname name of Epics PV (string)

param comparison operation used to compare PV value to trip_point.

type comparison string or callable. Built in comparisons are listed in *Table of Alarm Operators*.

param trip_point value that will trigger the alarm

param callback user-defined callback function to be run when the PVs value meets the alarm condition

type callback callable or None

param alert_delay time (in seconds) to wait before executing another alarm callback.

The alarm works by checking the value of the PV each time it changes. If the new value is outside the acceptable range (violates the trip point), then the user-supplied callback function is run. This callback could be set do send a message or to take some other course of action.

The comparison supplied can either be a string as listed in *Table of Alarm Operators* or a custom callable function which takes the two values (PV.value, trip_point) and returns `True` or `False` based on those values.

Table of built-in Operators for Alarms:

operator	Python operator
'eq', '=='	__eq__
'ne', '!='	__ne__
'le', '<='	__le__
'lt', '<'	__lt__
'ge', '>='	__ge__
'gt', '>'	__gt__

The `alert_delay` prevents the alarm callback from being called too many times. For PVs with floating point values, the value may fluctuate around the trip_point for a while. If the value violates the trip_point, then momentarily goes back to an acceptable value, and back again to a violating value, it may not be desirable to send repeated, identical messages. To prevent this situation, the alarm callback will be called when the alarm condition is met **and** the callback was not called within the time specified by `alert_delay`.

7.2 Alarm Example

An epics Alarm is very easy to use. Here is an alarm set to print a message when a PV's value reaches a certain value:

```
from epics import Alarm, poll

def alertMe(pvname=None, char_value=None, **kw):
    print "Soup's on!   %s = %s" % (pvname, char_value)

my_alarm = Alarm(pvname = 'WaterTemperature.VAL',
                 comparison = '>',
                 callback = alertMe,
                 trip_point = 100.0,
                 alert_delay = 600)
while True:
    poll()
```

CHAPTER EIGHT

AUTO-SAVING: SIMPLE SAVE/RESTORE OF PVS

The `autosave` module provides simple save/restore functionality for PVs, with the functions `save_pvs()` and `restore_pvs()`. These are similar to the autosave module from synApps for IOCs in that they use a compatible *request file* which describes the PVs to save, and a compatible *save file* which holds the saved values. Of course, the reading and writing is done here via Channel Access, and need not be related to any particular running IOC.

Use of this module requires the pyparsing package to be installed. This is a fairly common third-party python package, included in many package managers, or installed with tools such as *easy_install* or *pip*, or downloaded from PyPI

Request and Save file formats are designed to be compatible with synApps autosave. Notably, the *file* command with macro substitutions are supported, so that one can have a Request like:

```
# My.req
file "SimpleMotor.req", P=IOC:, Q=m1
```

with a **SimpleMotor.req** file of:

```
# SimpleMotor.req
$(P)$(Q).VAL
$(P)$(Q).DIR
$(P)$(Q).FOFF
```

which can then be used for many instances of a SimpleMotor. There is, however, no mechanism for automatically finding request files.

With such a file, simply using:

```
import epics.autosave
epics.autosave.save_pvs("My.req", "my_values.sav")
```

will save the current values for the PVs to the file **my_values.sav**. At a later time, these values can be restored with

> import epics.autosave epics.autosave.restore_pvs("my_values.sav")

The saved file will be of nearly identical format as that of the autosave mechanism, and the `restore_pvs()` function can read and restore values using save files from autosave. Note, however, that the purpose here is quite different from that of the standard autosave module (which is designed to save vales so that PVs can be **initialized** at IOC startup). Using the functions here will really do a `caput()` to the saved values.

`autosave.`**`save_pvs`**`(`*request_file*, *save_file*`)`

saves current value of PVs listed in *request_file* to the *save_file*

Parameters

- **`request_file`** – name of Request file to read PVs to save.
- **`save_file`** – name of file to save values to write values to

As discussed above, the **request_file** follows the conventions of the autosave module from synApps.

autosave.**restore_pvs**(*save_file*)

reads values from *save_file* and restores them for the corresponding PVs

Parameters **save_file** – name of file to save values to read data from.

Note that *restore_pvs()* will restore all the values it can, skipping over any values that it cannot restore.

8.1 Supported Data Types

All scalar PV values can be saved and restored with the *autosave* routines. There is some support for waveform (array) data. For example, character waveforms containing for long strings can be saved and restored. In addition, numerical arrays in waveform can be saved and restored. For array data, the results may not be fully compatible with the autosave module.

8.1.1 Examples

A simple example usign the autosave module:

```
import epics.autosave
# save values
epics.autosave.save_pvs("my_request_file.req",
                        "/tmp/my_recent_save.sav")

# wait 30 seconds
time.sleep(30)

# restore those values back
epics.autosave.restore_pvs("/tmp/my_recent_save.sav")
```

CHAPTER

NINE

WX: WXPYTHON WIDGETS FOR EPICS

The *wx* module of *epics* (that is, **epics.wx**) provides a set of wxPython classes for epics PVs. Most of these are derived from wxPython widgets, with special support added for epics PVs, especially regarding when to automatically update the widget based on a changing value for a PV.

Some examples of code that uses pyepics and wxPython are included in the *scripts* folder of the pyepics source distribution kit. In addition, there are a some full-fledged applications using Epics and wxPython at pyepics applications.

9.1 PV-aware Widgets

Several basic wxPython widgets have been extended so as to connect the widget with a corresponding *PV*. For example, setting the text value of a `PVTextCtrl` will forward that value to the epics *PV*, and if the *PV* is changed by externally, the value displayed in the widget will be automatically updated.

9.1.1 PVMixin

class `wx.`**`PVMixin`**(*[pv=None[, pvname=None]]*)
    This is a mixin class for wx Controls with epics PVs: This connects to PV, and manages connection and callback events for the PV. It provides the following basic methods used by most of the PV<->widget classes below.

`wx.`**`SetPV`**(*pv=None*)
    set the PV corresponding to the widget.

`wx.`**`Update`**(*value=None*)
    set the widgets value from the PV's value. If value=‘‘None‘‘, the current value for the PV is used.

`wx.`**`GetValue`**(*as_string=True*)
    return the PVs value.

`wx.`**`OnEpicsConnect`**()
    PV connection event handler.

`wx.`**`OnPVChange`**(*value*)
    PV monitor (subscription) event handler. Must be overwritten for each widget type.

`wx.`**`GetEnumStrings`**()
    return enumeration strings for the PV

9.1.2 PVCtrlMixin

class `wx.`**`PVCtrlMixin`**(*parent, pv=None, font=None, fg=None, bg=None, **kw*)

This is a mixin class for wx Controls with epics PVs: This subclasses PVCtrlMixin and adds colour translations PV, and manages callback events for the PV.

Parameters

- **`parent`** – wx parent widget
- **`pv`** – epics.PV
- **`font`** – wx.Font for display
- **`fg`** – foreground colour
- **`bg`** – background colour

A class that inherits from this class **must** provide a method called *_SetValue*, which will set the contents of the corresponding widget when the PV's value changes.

In general, the widgets will automatically update when the PV changes. Where appropriate, setting the value with the widget will set the PV value.

9.1.3 PVText

class `wx.`**`PVText`**(*parent*, *pv=None*, *font=None*, *fg=None*, *bg=None*, *minor_alarm="DARKRED"*, *major_alarm="RED"*, *invalid_alarm="ORANGERED"*, *auto_units=False*, *units=""*, ***kw*)

derived from wx.StaticText and PVCtrlMixin, this is a StaticText widget whose value is set to the string representation of the value for the corresponding PV.

By default, the text colour will be overridden when the PV enters an alarm state. These colours can be modified (or disabled by being set to `None`) as part of the constructor.

"units" specifies a unit suffix (like ' A' or ' mm') to put after the text value whenever it is displayed.

Alternatively, "auto_units" means the control will automatically display the "EGU" units value from the PV, whenever it updates. If this value is set, "units" is ignored. A space is inserted between the value and the unit.

9.1.4 PVTextCtrl

class `wx.`**`PVTextCtrl`**(*parent*, *pv=None*, *font=None*, *fg=None*, *bg=None*, *dirty_timeout=2500*, ***kw*)

derived from wx.TextCtrl and PVCtrlMixin, this is a TextCtrl widget whose value is set to the string representation of the value for the corresponding PV.

Setting the value (hitting Return or Enter) or changing focus away from the widget will set the PV value immediately. Otherwise, the widget will wait for 'dirty_timeout' milliseconds after the last keypress and then set the PV value to whatever is written in the field.

9.1.5 PVFloatCtrl

class `wx.`**`PVFloatCtrl`**(*parent*, *pv=None*, *font=None*, *fg=None*, *bg=None*, ***kw*)

A special variation of a wx.TextCtrl that allows only floating point numbers, as associated with a double, float, or integer PV. Trying to type in a non-numerical value will be ignored. Furthermore, if a PV's limits can be determined, they will be used to limit the allowed range of input values. For a value that is within limits, the value will be *put* to the PV on return. Out-of-limit values will be highlighted in a different color.

9.1.6 PVBitmap

class `wx.`**`PVBitmap`** (*parent*, *pv=None*, *bitmaps={}*, *defaultBitmap=None*)

A Static Bitmap where the image is based on PV value.

If the bitmaps dictionary is set, it should be set as PV.Value(Bitmap) where particular bitmaps will be shown if the PV takes those certain values.

If you need to do any more complex or dynamic drawing, you may want to look at the OGL PV controls.

9.1.7 PVCheckBox

class `wx.`**`PVCheckBox`** (*self*, *parent*, *pv=None*, *on_value=1*, *off_value=0*, ***kw*)

Checkbox based on a binary PV value, both reads/writes the PV on changes. on_value and off_value are the specific values that are mapped to the checkbox.

There are multiple options for translating PV values to checkbox settings (from least to most complex):

- Use a PV with values 0 and 1
- Use a PV with values that convert via Python's own bool(x)
- Set on_value and off_value in the constructor
- Use SetTranslations() to set a dictionary for converting various PV values to booleans.

9.1.8 PVFloatSpin

class `wx.`**`PVFloatSpin`** (*parent*, *pv=None*, *deadTime=500*, *min_val=None*, *max_val=None*, *increment=1.0*, *digits=-1*, ***kw*)

A FloatSpin is a floating point spin control with buttons to increase and decrease the value by a particular increment. Arrow keys and page up/down can also be used (the latter changes the value by 10x the increment.)

PVFloatSpin is a special derivation that assigns a PV to the FloatSpin control. deadTime is the delay (in milliseconds) between when the user finishes typing a value and when the PV is set to it (to prevent half-typed numeric values being set.)

9.1.9 PVButton

`PVButton(parent, pv=None, pushValue=1, disablePV=None, disableValue=1, **kw)`

A wx.Button linked to a PV. When the button is pressed, 'pushValue' is written to the PV (useful for momentary PVs with HIGH= set.) Setting disablePV and disableValue will automatically cause the button to disable when that PV has a certain value.

9.1.10 PVRadioButton

class `wx.`**`PVRadioButton`** (*parent*, *pv=None*, *pvValue=None*, ***kw*)

A PVRadioButton is a radio button associated with a particular PV and one particular value.

Suggested for use in a group where all radio buttons are PVRadioButtons, and they all have a discrete value set.

9.1.11 PVComboBox

class `wx.`**`PVComboBox`**(*parent*, *pv=None*, ***kw*)
A ComboBox linked to a PV. Both reads/writes the combo value on changes.

9.1.12 PVEnumComboBox

class `wx.`**`PVEnumComboBox`**(*parent*, *pv=None*, ***kw*)

A ComboBox linked to an "enum" type PV (such as bi,bo,mbbi,mbbo.) The ComboBox is automatically populated with a non-editable list of the PV enum values, allowing the user to select them from the drop-down.

Both reads/writes the combo value on changes.

9.1.13 PVEnumButtons

class `wx.`**`PVEnumButtons`**(*parent*, *pv=None*, *font=None*, *fg=None*, *bg=None*, ***kw*)
This will create a wx.Panel of buttons (a button bar), 1 for each enumeration state of an enum PV. The set of buttons will correspond to the current state of the PV

9.1.14 PVEnumChoice

class `wx.`**`PVEnumChoice`**(*parent*, *pv=None*, *font=None*, *fg=None*, *bg=None*, ***kw*)
This will create a dropdown list (a wx.Choice) with a list of enumeration states for an enum PV.

9.1.15 PVAlarm

class `wx.`**`PVAlarm`**(*parent*, *pv=None*, *font=None*, *fg=None*, *bg=None*, *trip_point=None*, ***kw*)
This will create a pop-up message (wx.MessageDialog) that is shown when the corresponding PV trips the alarm level.

9.1.16 PVCollapsiblePane

class `wx.`**`PVCollapsiblePane`**(*parent*, *pv=None*, *minor_alarm="DARKRED"*, *major_alarm="RED"*, *invalid_alarm="ORANGERED"*, ***kw*)
This is equivalent to wx.CollapsiblePane, except the label shown on the pane's "expansion button" comes from a PV.

The additional keyword arguments can be any of the other constructor arguments supported by wx.CollapsiblePane.

By default, the foreground colour of the pane button will be overridden when the PV enters an alarm state. On GTK, this means the colour of the triangular drop-down button but not the label text. These colours can be modified (or disabled by being set to `None`) as part of the constructor.

Supports the .SetTranslation() method, whose argument is a dictionary mapping PV values to display labels. If the PV value is not found in the dictionary, it will displayed verbatim as the label.

9.2 Decorators and other Utility Functions

`wx.DelayedEpicsCallback()`

decorator to wrap an Epics callback in a wx.CallAfter, so that the wx and epics ca threads do not clash This also checks for dead wxPython objects (say, from a closed window), and remove callbacks to them.

`wx.EpicsFunction()`

decorator to wrap function in a wx.CallAfter() so that Epics calls can be made in a separate thread, and asynchronously.

This decorator should be used for all code that mix calls to wx and epics

`wx.finalize_epics()`

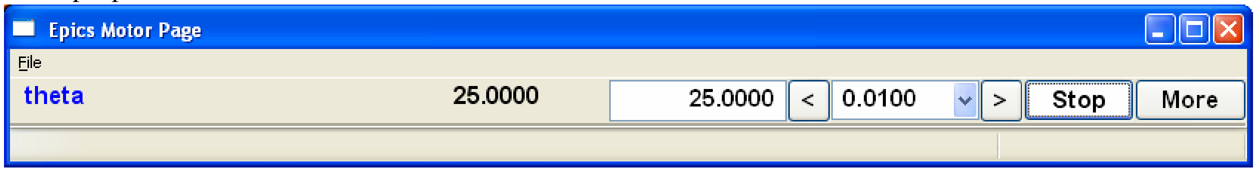
This function will finalize epics, and close all Channel Access communication, by calling `epics.ca.finalize_libca()`. This may be useful when closing an application, as in a method bound to *wx.EVT_CLOSE* event from a top-level application window. Be careful to **not** call this function when closing a Window if your application is not closing, and if you are still doing any Channel Access work in the other windows.

9.3 wxMotorPanel Widget

A dedicated wx Widget for Epics Motors is included in the `wx` module that provides an easy-to-use Motor panel that is similar to the normal MEDM window, but with a few niceties from the more sophisticated wx toolkit. This widget can be used simply as:

```
import wx
from epics.wx import MotorPanel
....
mymotor = MotorPanel(parent, 'XXX:m1')
```

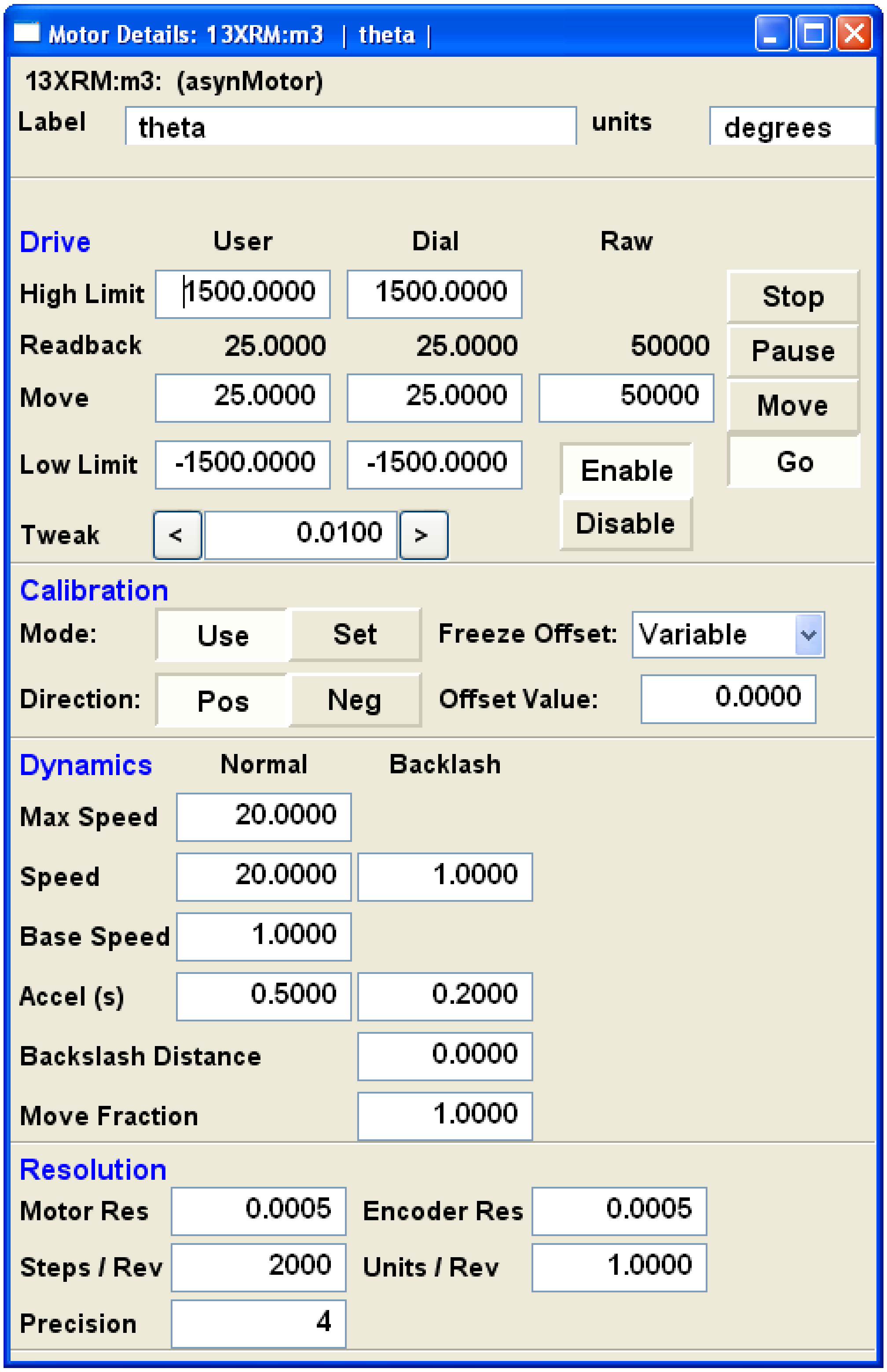
A sample panel looks like this



Which shows from right to left: the motor description, an information message (blank most of the time), the readback value, the drive value, arrows to tweak the motor, and a drop-down combobox for tweak values, a “Stop” button and a “More” button. The panel has the following features:

- All controls are “live” and will respond to changes from other source.
- The values for the tweak values in the ComboBox are automatically generated from the precision and travel range of the motor.
- The entry box for the drive value will *only* accept numeric input, and will only set the drive value when hitting Enter or Return.
- The drive value will change to Red text on a Yellow background when the value in the box violates the motors (user) limits. If Enter or Return when the the displayed value violates the limit, the motor will not be moved, but the displayed value will be changed to the closest limit value.
- Pressing the “Stop” button will stop the motor (with the *.SPMG* field), and set the Info field to “Stopped”. The button label will change to “Go”, and the motor will not move until this button is pressed.

Finally, the "More" button will bring up a more complete form of Motor parameters that looks like:



Motor Details: 13XRM:m3 | theta |

13XRM:m3: (asynMotor)

Label: theta
units: degrees

Drive	User	Dial	Raw
High Limit	1500.0000	1500.0000	
Readback	25.0000	25.0000	50000
Move	25.0000	25.0000	50000
Low Limit	-1500.0000	-1500.0000	
Tweak	0.0100		

Calibration

Mode: Use Set
Freeze Offset: Variable
Direction: Pos Neg
Offset Value: 0.0000

Dynamics	Normal	Backlash
Max Speed	20.0000	
Speed	20.0000	1.0000
Base Speed	1.0000	
Accel (s)	0.5000	0.2000
Backslash Distance		0.0000
Move Fraction		1.0000

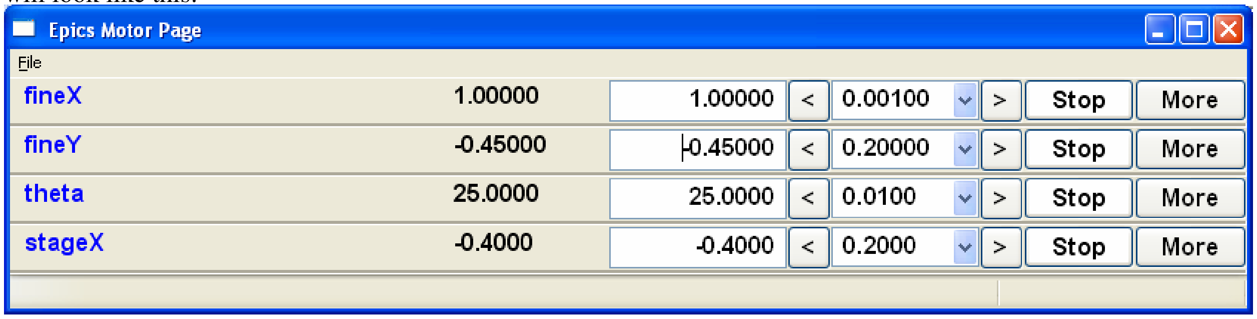
Resolution

Motor Res	0.0005	Encoder Res	0.0005
Steps / Rev	2000	Units / Rev	1.0000
Precision	4		

Many such MotorPanels can be put in a vertical stack, as generated from the 'wx_motor.py' script in the scripts folder of the source distribution as:

```
~>python wx_motor.py XXX:m1 XXX:m2 XXX:m3 XXX:m4
```

will look like this:



fineX	1.00000	1.00000	0.00100	
fineY	-0.45000	-0.45000	0.20000	
theta	25.0000	25.0000	0.0100	
stageX	-0.4000	-0.4000	0.2000	

9.4 OGL Classes

OGL is a graphics drawing library shipped with wxPython. Is it built around the concept of "shapes" which are added to "canvases" and can be moved, scrolled, zoomed, animated, etc.

There is a PVShapeMixin class which allows PV callback functionality to be added to any OGL Shape class, and there are also PVRectangle and PVCircle subclasses already created.

A recommended way to use these OGL classes is to make a static bitmap background for your display, place it in an OGL Canvas and then add an overlay of shapes which appear/disappear/resize/change colour based on the PV values.

9.4.1 PVShapeMixin

class `wx.`**`PVShapeMixin`** (*self*, *pv=None*, *pvname=None*)

Similar to PVMixin, this mixin should be added to any ogl.Shape subclass that needs PV callback support.

The main method is PVChanged(self, raw_value), which should be overridden in the subclass to provide specific processing based on the changed value.

There are also some built-in pieces of functionality. These are enough to do simple show/hide or change colour shape functionality, without needing to write specific code.

SetBrushTranslations(translations) allows setting a dict of PV Value -> wx.Brush mappings, which can be used to automatically repaint the shape foreground (fill) when the PV changes.

SetPenTranslations(translations) similar to brush translations, but the values are wx.Pen instances that are used to repaint the shape outline when the PV changes.

SetShownTranslations(translations) sets a dictionary of PV Value ->bool values which are used to show/hide the shape depending on the PV value, as it changes.

9.4.2 PVRectangle

class `wx.`**`PVRectangle`** (*self*, *w*, *h*, *pv=None*, *pvname=None*)

A PVCtrlMixin for the Rectangle shape class.

9.4.3 PVCircle

class `wx.`**`PVCircle`**(*self*, *diameter*, *pv=None*, *pvname=None*)

A PVCtrlMixin for the Circle shape class.

CHAPTER

TEN

ADVANCED TOPIC WITH PYTHON CHANNEL ACCESS

This chapter contains a variety of "usage notes" and implementation details that may help in getting the best performance from the pyepics module.

10.1 The wait and timeout options for get(), ca.get_complete()

The *get* functions, `epics.caget()`, `pv.get()` and `ca.get()` all ask for data to be transferred over the network. For large data arrays or slow networks, this can can take a noticeable amount of time. For PVs that have been disconnected, the *get* call will fail to return a value at all. For this reason, these functions all take a *timeout* keyword option. The lowest level `ca.get()` also has a *wait* option, and a companion function `ca.get_complete()`. This section describes the details of these.

If you're using `epics.caget()` or `pv.get()` you can supply a timeout value. If the value returned is `None`, then either the PV has truly disconnected or the timeout passed before receiving the value. If the *get* is incomplete, in that the PV is connected but the data has simply not been received yet, a subsequent `epics.caget()` or `pv.get()` will eventually complete and receive the value. That is, if a PV for a large waveform record reports that it is connected, but a `pv.get()` returns None, simply trying again later will probably work:

```
>>> p = epics.PV('LargeWaveform')
>>> val = p.get()
>>> val
>>> time.sleep(10)
>>> val = p.get()
```

At the lowest level (which `pv.get()` and `epics.caget()` use), `ca.get()` issues a get-request with an internal callback function. That is, it calls the CA library function `libca.ca_array_get_callback()` with a predefined callback function. With *wait=True* (the default), `ca.get()` then waits up to the timeout or until the CA library calls the specified callback function. If the callback has been called, the value can then be converted and returned.

If the callback is not called in time or if *wait=False* is used but the PV is connected, the callback will be called eventually, and simply waiting (or using `ca.pend_event()` if `ca.PREEMPTIVE_CALLBACK` is `False`) may be sufficient for the data to arrive. Under this condition, you can call `ca.get_complete()`, which will NOT issue a new request for data to be sent, but wait (for up to a timeout time) for the previous get request to complete.

`ca.get_complete()` will return `None` if the timeout is exceeded or if there is not an "incomplete get" that it can wait to complete. Thus, you should use the return value from `ca.get_complete()` with care.

Note that `pv.get()` (and so `epics.caget()`) will normally rely on the PV value to be filled in automatically by monitor callbacks. If monitor callbacks are disabled (as is done for large arrays and can be turned off) or if the monitor hasn't been called yet, `pv.get()` will check whether it should can `ca.get()` or `ca.get_complete()`.

If not specified, the timeout for `ca.get_complete()` (and all other get functions) will be set to:

```
timeout = 0.5 + log10(count)
```

Again, that's the maximum time that will be waited, and if the data is received faster than that, the *get* will return as soon as it can.

10.2 Strategies for connecting to a large number of PVs

Occasionally, you may find that you need to quickly connect to a large number of PVs, say to write values to disk. The most straightforward way to do this, say:

```
import epics

pvnamelist = read_list_pvs()
pv_vals = {}
for name in pvnamelist:
    pv = epics.PV(name)
    pv_vals[name] = pv.get()
```

does incur some small performance penalty. As shown below, the penalty is generally pretty small in absolute terms, but can be noticeable when you are connecting to a large number (say, more than 100) PVs at once.

The cause for the penalty, and its remedy, are two-fold. First, a *PV* object automatically use connection and event callbacks. Normally, these are advantages, as you don't need to explicitly deal with them. But, internally, they do pause for network responses using `ca.pend_event()` and these pauses can add up. Second, the `ca.get()` also pauses for network response, so that the returned value actually contains the latest data right away, as discussed in the previous section.

The remedies are to

1. not use connection or event callbacks.
2. not explicitly wait for values to be returned for each `get()`.

A more complicated but faster approach relies on a carefully-tuned use of the CA library, and would be the following:

```
from epics import ca

pvnamelist = read_list_pvs()

pvdata = {}
for name in pvnamelist:
    chid = ca.create_channel(name, connect=False, auto_cb=False) # note 1
    pvdata[name] = (chid, None)

for name, data in pvdata.items():
    ca.connect_channel(data[0])
ca.poll()
for name, data in pvdata.items():
    ca.get(data[0], wait=False)  # note 2

ca.poll()
for name, data in pvdata.items():
    val = ca.get_complete(data[0])
    pvdata[name][1] = val

for name, data in pvdata.items():
    print name, data[1]
```

The code here probably needs detailed explanation. The first thing to notice is that this is using the *ca* level, not *PV* objects. Second (Note 1), the *connect=False* and *auto_cb=False* options to `ca.create_channel()`. These respectively tell `ca.create_channel()` to not wait for a connection before returning, and to not automatically assign a connection callback. Normally, these are not what you want, as you want a connected channel and to know if the connection state changes. But we're aiming for maximum speed here, so we avoid these.

We then explicitly call `ca.connect_channel()` for all the channels. Next (Note 2), we tell the CA library to request the data for the channel without waiting around to receive it. The main point of not having `ca.get()` wait for the data for each channel as we go is that each data transfer takes time. Instead we request data to be sent in a separate thread for all channels without waiting. Then we do wait by calling `ca.poll()` once and only once, (not len(channels) times!). Finally, we use the `ca.get_complete()` method to convert the data that has now been received by the companion thread to a python value.

How much faster is the more explicit method? In my tests, I used 20,000 PVs, all scalar values, all actually connected, and all on the same subnet as the test client, though on a mixture of several vxWorks and linux IOCs. I found that the simplest, obvious approach as above took around 12 seconds to read all 20,000 PVs. Using the *ca* layer with connection callbacks and a normal call to `ca.get()` also took about 12 seconds. The method without connection callbacks and with delayed unpacking above took about 2 seconds to read all 20,000 PVs.

Is that performance boost from 12 to 2 seconds significant? If you're writing a script that is intended to run once, fetch a large number of PVs and get their values (say, an auto-save script that runs on demand), then the boost is definitely significant. On the other hand, if you're writing a long running process or a process that will retain the PV connections and get their values multiple times, the difference in start-up speed is less significant. For a long running auto-save script that periodically writes out all the PV values, the "obvious" way using automatically monitored PVs may be much *better*, as the time for the initial connection is small, and the use of event callbacks will reduce network traffic for PVs that don't change between writes.

Note that the tests also show that, with the simplest approach, 1,000 PVs should connect and receive values in under 1 second. Any application that is sure it needs to connect to PVs faster than that rate will want to do careful timing tests. Finally, note also that the issues are not really a classic *python is slow compared to C* issue, but rather a matter of how much pausing with `ca.poll()` one does to make sure values are immediately useful.

10.3 time.sleep() or epics.poll()?

In order for a program to communicate with Epics devices, it needs to allow some time for this communication to happen. With `ca.PREEMPTIVE_CALLBACK` set to `True`, this communication will be handled in a thread separate from the main Python thread. This means that CA events can happen at any time, and `ca.pend_event()` does not need to be called to explicitly allow for event processing.

Still, some time must be released from the main Python thread on occasion in order for events to be processed. The simplest way to do this is with `time.sleep()`, so that an event loop can simply be:

```
>>> while True:
>>>     time.sleep(0.001)
```

Unfortunately, the `time.sleep()` method is not a very high-resolution clock, with typical resolutions of 1 to 10 ms, depending on the system. Thus, even though events will be asynchronously generated and epics with pre-emptive callbacks does not *require* `ca.pend_event()` or `ca.poll()` to be run, better performance may be achieved with an event loop of:

```
>>> while True:
>>>     epics.poll(evt=1.e-5, iot=0.1)
```

as the loop will be run more often than using `time.sleep()`.

10.4 Using Python Threads

An important feature of the PyEpics package is that it can be used with Python threads, as Epics 3.14 supports threads for client code. Even in the best of cases, working with threads can be somewhat tricky and lead to unexpected behavior, and the Channel Access library adds a small level of complication for using CA with Python threads. The result is that some precautions may be in order when using PyEpics and threads. This section discusses the strategies for using threads with PyEpics.

First, to use threads with Channel Access, you must have `epics.ca.PREEMPTIVE_CALLBACK` = `True`. This is the default value, but if `epics.ca.PREEMPTIVE_CALLBACK` has been set to `False`, threading will not work.

Second, if you are using `PV` objects and not making heavy use of the `ca` module (that is, not making and passing around chids), then the complications below are mostly hidden from you. If you're writing threaded code, it's probably a good idea to read this just to understand what the issues are.

10.4.1 Channel Access Contexts

The Channel Access library uses a concept of *contexts* for its own thread model, with contexts holding sets of threads as well as Channels and Process Variables. For non-threaded work, a process will use a single context that is initialized prior doing any real CA work (done in `ca.initialize_libca()`). In a threaded application, each new thread begins with a new, uninitialized context that must be initialized or replaced. Thus each new python thread that will interact with CA must either explicitly create its own context with `ca.create_context()` (and then, being a good citizen, destroy this context as the thread ends with `ca.destroy_context()`) or attach to an existing context.

The generally recommended approach is to use a single CA context throughout an entire process and have each thread attach to the first context created (probably from the main thread). This avoids many potential pitfalls (and crashes), and can be done fairly simply. It is the default mode when using PV objects.

The most explicit use of contexts is to put `epics.ca.create_context()` at the start of each function call as a thread target, and `epics.ca.destroy_context()` at the end of each thread. This will cause all the activity in that thread to be done in its own context. This works, but means more care is needed, and so is not the recommended.

The best way to attach to the initially created context is to call `epics.ca.use_initial_context()` before any other CA calls in each function that will be called by `Thread.run()`. Equivalently, you can add a `withInitialContext()` decorator to the function. Creating a PV object will implicitly do this for you, as long as it is your first CA action in the function. Each time you do a `PV.get()` or `PV.put()` (or a few other methods), it will also check that the initial context is being used.

Of course, this approach requires CA to be initialized already. Doing that *in the main thread* is highly recommended. If it happens in a child thread, that thread must exist for all CA work, so either the life of the process or with great care for processes that do only some CA calls. If you are writing a threaded application in which the first real CA calls are inside a child thread, it is recommended that you initialize CA in the main thread,

As a convenience, the `CAThread` in the `ca` module is is a very thin wrapper around the standard `threading.Thread` which adding a call of `epics.ca.use_initial_context()` just before your threaded function is run. This allows your target functions to not explicitly set the context, but still ensures that the initial context is used in all functions.

10.4.2 How to work with CA and Threads

Summarizing the discussion above, to use threads you must use run in PREEMPTIVE_CALLBACK mode. Furthermore, it is recommended that you use a single context, and that you initialize CA in the main program thread so that your single CA context belongs to the main thread. Using PV objects exclusively makes this easy, but it can also be accomplished relatively easily using the lower-level ca interface. The options for using threads (in approximate order of reliability) are then:

1. use PV objects for threading work. This ensures you're working in a single CA context.
2. use `CAThread` instead of `Thread` for threads that will use CA calls.
3. put `epics.ca.use_initial_context()` at the top of all functions that might be a Thread target function, or decorate them with `withInitialContext()` decorator, *@withInitialContext*.
4. use `epics.ca.create_context()` at the top of all functions that are inside a new thread, and be sure to put `epics.ca.destroy_context()` at the end of the function.
5. ignore this advise and hope for the best. If you're not creating new PVs and only reading values of PVs created in the main thread inside a child thread, you may not see a problems, at least not until you try to do something fancier.

10.4.3 Thread Examples

This is a simplified version of test code using Python threads. It is based on code originally from Friedrich Schotte, NIH, and included as *thread_test.py* in the *tests* directory of the source distribution.

In this example, we define a *run_test* procedure which will create PVs from a supplied list, and monitor these PVs, printing out the values when they change. Two threads are created and run concurrently, with overlapping PV lists, though one thread is run for a shorter time than the other.

```python
"""This script tests using EPICS CA and Python threads together
Based on code from  Friedrich Schotte, NIH, modified by Matt Newville
19-Apr-2010
"""
import time
from  sys import stdout
from threading import Thread
import epics
from epics.ca import CAThread

from  pvnames import updating_pvlist
pvlist_a = updating_pvlist[:-1]
pvlist_b = updating_pvlist[1:]

def run_test(runtime=1, pvnames=None,  run_name='thread c'):
    msg = '-> thread "%s" will run for %.3f sec, monitoring %s\n'
    stdout.write(msg % (run_name, runtime, pvnames))
    def onChanges(pvname=None, value=None, char_value=None, **kw):
        stdout.write('   %s = %s (%s)\n' % (pvname, char_value, run_name))
        stdout.flush()

    # epics.ca.use_initial_context()   #  epics.ca.create_context()
    start_time = time.time()
    pvs = [epics.PV(pvn, callback=onChanges) for pvn in pvnames]

    while time.time()-start_time < runtime:
        time.sleep(0.1)

    [p.clear_callbacks() for p in pvs]
    stdout.write( 'Completed Thread  %s\n' % ( run_name))

stdout.write( "First, create a PV in the main thread:\n")
p = epics.PV(updating_pvlist[0])

stdout.write("Run 2 Background Threads simultaneously:\n")
th1 = CAThread(target=run_test,args=(3, pvlist_a,  'A'))
```

```
th1.start()

th2 = CAThread(target=run_test,args=(6, pvlist_b, 'B'))
th2.start()

th2.join()
th1.join()
stdout.write('Done\n')
```

In light of the long discussion above, a few remarks are in order: This code uses the standard Thread library and explicitly calls *epics.ca.use_initial_context()* prior to any CA calls in the target function. Also note that the `run_test()` function is first called from the main thread, so that the initial CA context does belong to the main thread. Finally, the *epics.ca.use_initial_context()* call in `run_test()` above could be replaced with *epics.ca.create_context()*, and run OK.

The output from this will look like:

```
First, create a PV in the main thread:
Run 2 Background Threads simultaneously:
-> thread "A" will run for 3.000 sec, monitoring ['Py:ao1', 'Py:ai1', 'Py:long1']
-> thread "B" will run for 6.000 sec, monitoring ['Py:ai1', 'Py:long1', 'Py:ao2']
   Py:ao1 = 8.3948 (A)
   Py:ai1 = 3.14 (B)
   Py:ai1 = 3.14 (A)
   Py:ao1 = 0.7404 (A)
   Py:ai1 = 4.07 (B)
   Py:ai1 = 4.07 (A)
   Py:long1 = 3 (B)
   Py:long1 = 3 (A)
   Py:ao1 = 13.0861 (A)
   Py:ai1 = 8.49 (B)
   Py:ai1 = 8.49 (A)
   Py:ao2 = 30 (B)
Completed Thread  A
   Py:ai1 = 9.42 (B)
   Py:ao2 = 30 (B)
   Py:long1 = 4 (B)
   Py:ai1 = 3.35 (B)
   Py:ao2 = 31 (B)
   Py:ai1 = 4.27 (B)
   Py:ao2 = 31 (B)
   Py:long1 = 5 (B)
   Py:ai1 = 8.20 (B)
   Py:ao2 = 31 (B)
Completed Thread  B
Done
```

Note that while both threads *A* and *B* are running, a callback for the PV *Py:ai1* is generated in each thread.

Note also that the callbacks for the PVs created in each thread are **explicitly cleared** with:

```
[p.clear_callbacks() for p in pvs]
```

Without this, the callbacks for thread *A* will persist even after the thread has completed!

10.5 Using Multiprocessing with PyEpics

An alternative to Python threads that has some very interesting and important features is to use multiple *processes*, as with the standard Python `multiprocessing` module. While using multiple processes has some advantages over threads, it also has important implications for use with PyEpics. The basic issue is that multiple processes need to be fully separate, and do not share global state. For epics Channel Access, this means that all those things like established communication channels, callbacks, and Channel Access **context** cannot easily be share between processes.

The solution is to use a *`CAProcess`*, which acts just like `multiprocessing.Process`, but knows how to separate contexts between processes. This means that you will have to create PV objects for each process (even if they point to the same PV).

class `CAProcess` (*group=None*, *target=None*, *name=None*, *args=()*, *kwargs={}*)
a subclass of `multiprocessing.Process` that clears the global Channel Access context before running you target function in its own process.

class `CAPool` (*processes=None*, *initializer=None*, *initargs=()*, *maxtasksperchild=None*)
a subclass of `multiprocessing.pool.Pool`, creating a Pool of *`CAProcess`* instances.

A simple example of using multiprocessing successfully is given:

```
from __future__ import print_function
import epics
import time
import multiprocessing as mp
import threading

import pvnames
PVN1 = pvnames.double_pv # 'Py:ao2'
PVN2 = pvnames.double_pv2 # 'Py:ao3'

def subprocess(*args):
    print('==subprocess==', args)
    mypvs = [epics.PV(pvname) for pvname in args]

    for i in range(10):
        time.sleep(0.750)
        out = [(p.pvname, p.get(as_string=True)) for p in mypvs]
        out = ', '.join(["%s=%s" % o for o in out])
        print('==sub (%d): %s' % (i, out))

def main_process():
    def monitor(pvname=None, char_value=None, **kwargs):
        print('--main:monitor %s=%s' % (pvname, char_value))

    print('--main:')
    pv1 = epics.PV(PVN1)
    print('--main:init %s=%s' % (PVN1, pv1.get()))
    pv1.add_callback(callback=monitor)

    try:
        proc1 = epics.CAProcess(target=subprocess,
                                args=(PVN1, PVN2))
        proc1.start()
        proc1.join()
    except KeyboardInterrupt:
        print('--main: killing subprocess')
        proc1.terminate()
    print('--main: subprocess complete')
    time.sleep(0.9)
    print('--main:final %s=%s' % (PVN1, pv1.get()))

if __name__ == '__main__':
    main_process()
```

here, the main process and the subprocess can each interact with the same PV, though they need to create a separate connection (here, using `PV`) in each process.

Note that different *CAProcess* instances can communicate via standard `multiprocessing.Queue`. At this writing, no testing has been done on using multiprocessing Managers.