pcaspy Documentation

Release 0.6.5

Xiaoqiang Wang

May 10, 2017

Contents

1 Overview **1**

2 Contents **3**

2.1 Installation 3
2.2 Tutorial 6
2.3 Other Tips 14
2.4 Reference 16
2.5 Development 20
2.6 Release Notes 23

3 Indices and tables **27**

Python Module Index **29**

CHAPTER 1

Overview

PCASpy provides not only the low level python binding to EPICS Portable Channel Access Server but also the necessary high level abstraction to ease the server tool programming.

To get PCASpy for your system, checkout the *Installation* guide. Then to get started with, check out a series of *Tutorial*. It walks through the principles of a PCASpy application. After that you should feel confident to start your adventure. If necessary consult the *Reference* about the API.

After you have created an application, be it generic or site specific, share your experience at success stories and let others be inspired.

CHAPTER 2

Contents

Installation

Binary Installers

Anaconda

Packages for Anaconda can be installed via:

```
conda install -c paulscherrerinstitute pcaspy
```

Wheel

The binary packages are distributed at PyPI. They have EPICS 3.14.12.6 libraries statically builtin. Make sure you have pip and wheel installed, and run:

```
$ sudo pip install pcaspy # macOS
> C:\Python27\Scripts\pip.exe install pcaspy :: Windows
```

Note: On Windows, if you see error message "The program can't start because MSVCRxxx.dll is missing from your computer." when importing pcaspy, you might need to install the proper Visual C++ Redistributable.

Egg

PyPI does not allow upload linux-specific wheels package, yet (as of 2014). The old *egg* format is used then:

```
$ sudo easy_install pcaspy
```

Or install only for the current user:

```
$ easy_install --user pcaspy
```

Source

If no binary package is available for your system, you can build from source. And then you would need EPICS base installed, see *Getting EPICS*.

The source can be downloaded in various ways:

- The released source tarballs can be found at PyPI.
- From the git repository, each release can be downloaded as a zip package.
- Clone the repository if you feel adventurous:

```
$ git clone https://github.com/paulscherrerinstitute/pcaspy.git
```

Getting EPICS

In general please follow the official installation instruction. Here is a short guide,

- Get the source tarball from http://www.aps.anl.gov/epics/base/R3-14/12.php.
- Unpack it to a proper path.
- Set the following environment variables:
  - EPICS_BASE : the path containing the EPICS base source tree.
  - EPICS_HOST_ARCH : EPICS is built into static libraries on Windows.

OS	Arch	EPICS_HOST_ARCH
Linux	32bit	linux-x86
	64bit	linux-x86_64
Windows	32bit	win32-x86-static
	64bit	windows-x64-static
OS X	PPC	darwin-ppcx86
	Intel	darwin-x86

- Run `make`.

Note: On windows, the Visual Studio version has to match that used to build Python.

Python Version	Visual Studio Version
2.6 - 2.7, 3.0 - 3.2	2008
3.3 - 3.4	2010
3.5 - 3.6	2015

Mismatching may cause crashes!

Windows

- Python 2.6+ including 3.x (http://www.python.org/download/)
- SWIG 1.3.29+ is required. Get it from http://www.swig.org/download.html and unpack to `C:\Program Files (x86)\SWIG\`.

Download the most recent source tarball, uncompress and run:

```
> set PATH=%PATH%;C:\Program Files (x86)\SWIG\
> C:\Python27\python.exe setup.py build install
```

Linux

- Python 2.6+ including 3.x
- Python headers (package name "python-dev" or similar)
- SWIG 1.3.29+ (package name "swig")

In the source directory, run:

```
$ sudo python setup.py install
```

or install only for the current user:

```
$ python setup.py build install --user
```

Note: You might need to pass *-E* flag to sudo to preserve the EPICS environment variables. If your user account is not allowed to do so, a normal procedure should be followed,

```
$ su -
# export EPICS_BASE=<epics base path>
# export EPICS_HOST_ARCH=<epics host arch>
# python setup.py install
```

---

macOS

- SWIG (MacPorts package "swig-python")

In the source directory, run:

```
$ sudo python setup.py install
```

Package

After the build succeeds, you may want to create a package for distribution.

Anaconda

Conda recipe is included:

```
$ conda build -c paulscherrerinstitute conda-recipe
```

Wheel

```
$ python setup.py bdist_wheel
```

RPM

The spec file *python-pcaspy.spec* is included. Get the source tarball either from PyPI or create it by `python setup.py sdist`, and run:

```
$ rpmbuild -ta pcaspy-0.6.3.tar.gz
```

The binary and source RPM will be created. The package name is *python-pcaspy*.

Tutorial

Example 1: Expose some random number(s)

Define PV database

Suppose we want to have one PV returning a random number, we define it like this:

```
prefix = 'MTEST:'
pvdb = {
    'RAND' : {
        'prec' : 3,
    },
}
```

`pvdb` is a plain Python `dict`, in which keys are PV base names and values are their configurations, a dict also. In this case we specify that the PV has base name `RAND` and 3 digits precision. `prefix` will be the prefixed to the PV base names and create the PV `MTEST:RAND`.

Refer to *Database Field Definition* about full description of database definition.

Dummy server

Here is the source code dummy.py.

Our first attempt is to implement a server so it blindly accepts any value written and gives it back upon request. Two classes are required,:

```
from pcaspy import SimpleServer, Driver
```

`Driver` class is the base class that connects channel access requests with real world data source. The base class implementation simply stores the value written by user and retrieves upon request. All the derived class does is to call base class's *`Driver.__init__()`* to ensure proper setup.:

```
class myDriver(Driver):
    def __init__(self):
        super(myDriver, self).__init__()
```

SimpleServer is the class that responds to channel access requests. We would never need to modify it. We only need to instantiate it.:

```
if __name__ == '__main__':
    server = SimpleServer()
```

And create all PVs based on `prefix` and `pvdb` definition.:

```
server.createPV(prefix, pvdb)
driver = myDriver()
```

In the end we start the processing loop.:

```
while True:
    # process CA transactions
    server.process(0.1)
```

Now try some caget/caput,:

```
$ caput MTEST:RAND 0
Old : MTEST:RAND                            2
New : MTEST:RAND                            0
$ caput MTEST:RAND -1.23
Old : MTEST:RAND                            0
New : MTEST:RAND                            -1.23
$ caget MTEST:RAND
MTEST:RAND                          -1.23
```

Notice that the procedure to instantiate server, PVs and driver remains identical afterwards. So later we will not show this part.

Return a random number

It is not much interesting to be just an echo. We would return a random number upon every read request. We need to override *read* method in our subclass *myDriver*.:

```
import random
class myDriver(Driver):
    def __init__(self):
        super(myDriver, self).__init__()

    def read(self, reason):
        if reason == 'RAND':
            value = random.random()
        else:
            value = self.getParam(reason)
        return value
```

Note:

- *Driver.__init__()* must be called **before** using any *Driver.getParam()* *Driver.setParam()* calls.
- The `read` method accepts one parameter `reason` and it is the PV base name as defined in `pvdb`. In this function, we return a random number when `RAND` is being read.

Scan periodically

Until now this PV updates only when clients read. It can also update itself periodically if we define the *scan* field,:

```
pvdb = {
    'RAND' : {
        'prec' : 3,
        'scan' : 1,
    },
}
```

Now the PV will update every second. Monitor the change,:

```
$ camonitor MTEST:RAND
MTEST:RAND                     2011-07-19 12:32:06.574775 0.646198
MTEST:RAND                     2011-07-19 12:32:07.574704 0.872313
MTEST:RAND                     2011-07-19 12:32:08.581681 0.171537
MTEST:RAND                     2011-07-19 12:32:09.581581 0.351235
```

Note:

- The scan thread is implemented per PV and the scan interval can be arbitrary. In comparison the EPICS database scan thread is implemented per IOC and scan interval is defined in database definition.

Return a series of random number

Suppose we want to return more random numbers, 10 e.g. Add *count* field in `pvdb`,:

```
pvdb = {
    'RAND' : {
        'prec' : 3,
        'scan' : 1,
        'count': 10,
    },
}
```

We modify the `read` method to return a list of 10 random numbers,:

```
...
if reason == 'RAND':
    return [random.random() for i in range(10)]
...
```

Here is the final source code get_random.py

Remark

This first demo shows the basics of how to configure PV attributes and respond to read access. One thing to emphasise is that *Driver.read()* is called each time a ca_get request comes in. In a realistic application, the PV values are normally polled, in a periodical or triggered way, from external sources. As so the PV values will be stored in a parameter cache (*Driver.setParam()*) at the point of being fetched. The derived driver does not need to override *Driver.read()*. The values are simply fetched from the parameter cache (*Driver.getParam()*). This principle is followed by all the following examples.

Example 2: Interface to any shell command

Here is the full source code pysh.py

Define PV database

Suppose we want to run a shell command and publish its results as an EPICS channel.:

```
prefix = 'MTEST:'
pvdb = {
    'COMMAND' : {
        'type' : 'string',
    },
    'OUTPUT'  : {
        'type' : 'string',
    },
    'STATUS'  : {
        'type' : 'enum',
        'enums': ['DONE', 'BUSY']
    },
    'ERROR'   : {
        'type' : 'string',
    },
}
```

`MTEST:COMMAND` contains the command to execute. Its output will be stored in `MTEST:OUTPUT`. Its possible error will be stored in `MTEST:ERROR`. `MTEST:STATUS` indicates whether the command finishes running or not.

Implement myDriver

We do the normal inheritance of `Driver`,:

```
import thread
import subprocess
import shlex

from pcaspy import Driver, SimpleServer

class myDriver(Driver):
    def __init__(self, server):
        Driver.__init__(self, server)
        # shell execution thread id
        self.tid = None
```

In this driver, readout is done through the default implementation of Driver, which retrieves the value with *Driver.getParam()*. So we will not override *Driver.read()*.

`write` method

In the `write` method, we only respond to the write request of `MTEST:COMMAND`. If there is no command running, we spawn a new thread to run the command in `runShell`.:

```
def write(self, reason, value):
    status = True
    if reason == 'COMMAND':
```

```
        if not self.tid:
            command = value
            self.tid = thread.start_new_thread(self.runShell,(command,))
        else:
            status = False
    else:
        status = False
    # store the values
    if status:
        self.setParam(reason, value)

    return status
```

Note:

- Portable channel access server is single threaded so we should avoid blocking the `write` method by any means. In this case we run the command in a new thread.
- We have limited the running command to one. Until the running thread finishes, `status = False` is returned to refuse further requests and the client may see a put failure.
- We assign `status = False` to refuse change requests of `OUTPUT`, `ERROR` and `STATUS`. This makes them effectively read-only.

Execution thread

In our command execution thread, we run the command with `subprocess` module. The subprocess's *stdout* and *stderr* outputs are redirected to channel `MTEST:OUTPUT` and `MTEST:ERROR`. Upon exception `MTEST:ERROR` has the exception message.

Before and after command execution we update `MTEST:STATUS` channel. We call *`Driver.updatePVs()`* to inform clients about PV value change.:

```
def runShell(self, command):
    # set status BUSY
    self.setParam('STATUS', 1)
    self.updatePVs()
    # run shell
    try:
        proc = subprocess.Popen(shlex.split(command),
                stdout = subprocess.PIPE,
                stderr = subprocess.PIPE)
        proc.wait()
    except OSError, m:
        self.setParam('ERROR', str(m))
        self.setParam('OUTPUT', '')
    else:
        self.setParam('ERROR', proc.stderr.read().rstrip())
        self.setParam('OUTPUT', proc.stdout.read().rstrip())
    # set status DONE
    self.setParam('STATUS', 0)
    self.updatePVs()
    self.tid = None
```

Now we can run some commands to see the output,:

```
$ caput MTEST:COMMAND "whoami"
Old : MTEST:COMMAND
New : MTEST:COMMAND            whoami
$ caget MTEST:OUTPUT
MTEST:OUTPUT                   wang_x1
```

Make it asynchronous

As we have noted, the command normally would take undetermined time to finish running. In addiction to yield `MTEST:STATUS` to indicate completion. We could make `MTEST:COMMAND` asynchronous, and notify upon completion if client has called *ca_array_put_callback*.

Add a new field *asyn* to `COMMAND` to indicate that this PV finishes writing asynchronously,:

```
'COMMAND' : {
    'type' : 'string',
    'asyn' : True
},
```

In thread *runShell*, we call *`Driver.callbackPV()`* to notify the processing is done.:

```
# run shell
...
self.callbackPV('COMMAND')
# set status DONE
```

Now run it again and notice the delay,:

```
$ caput -w 10 -c MTEST:COMMAND "sleep 5"
Old : MTEST:COMMAND                whoami
New : MTEST:COMMAND                sleep 5
```

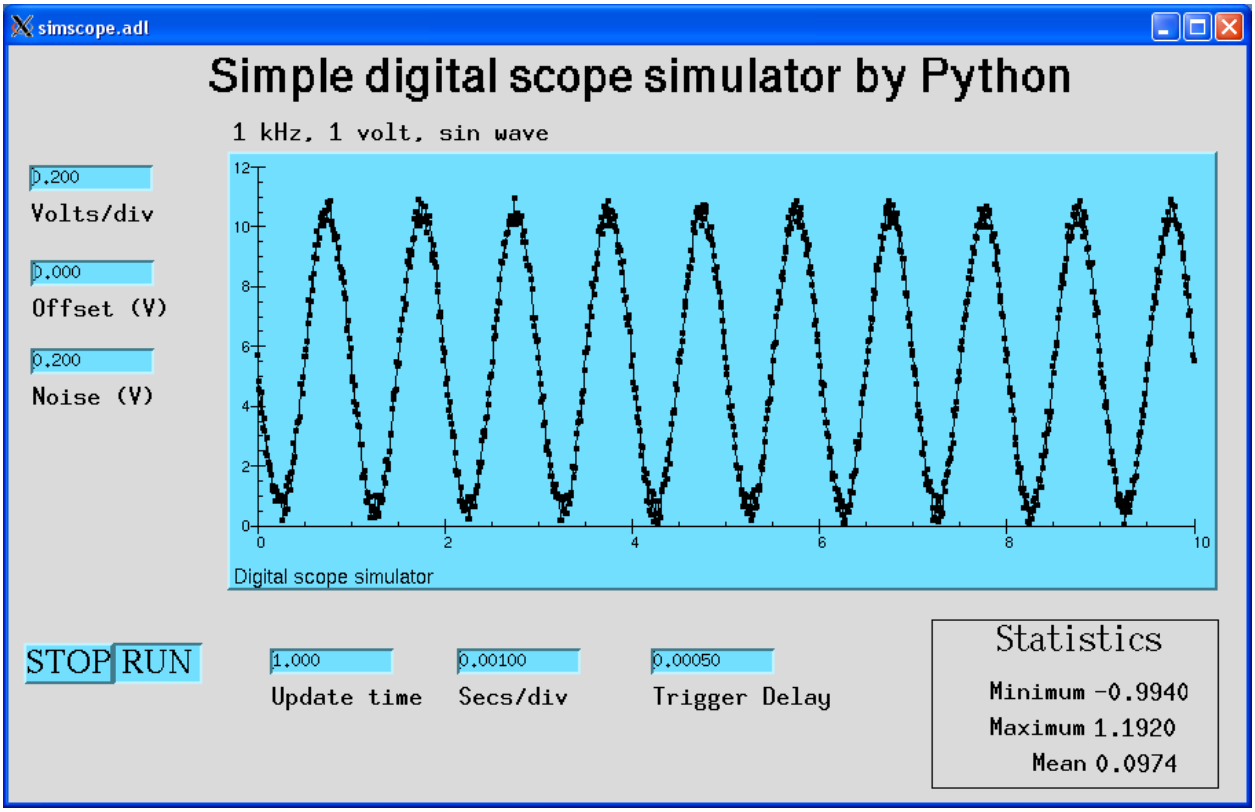
Example 3: A Simulated Oscilloscope

Until now the driver principles have all been introduced by these two trivial examples. I cannot find more realistic examples, so I port this asynPortDriver example. Its intention is stated clearly by Mark Rivers,

> This example is a simple digital oscilloscope emulator. In this example all of output control and input data is done in a calculated simulation. But it is easy to see how to use the driver as a basis for real device control. The code doing the simulation would simply be changed to talk to an actual device.

The python version in PCASpy is simscope.py. To best check how it functions, launch the medm panel,:

```
medm -x -macro P=MTEST simscope.adl
```



Note:

- The value passed to `setParam` could be Python builtin types: str, float, int, list, tuple or numpy data types: int8/16/32, float32/64, ndarray.

Example 4: Integrate into GUI applications

In the above examples, the server process loop is running in the main thread. GUI applications require their own event loop running in the main thread also. In such application the server process loop could run in a separate thread and yield the main thread to the GUI event loop.

A helper class *`ServerThread`* can be used to execute the server in a separate thread.

The following runs the server for ~4 seconds and exits. The debug output shows the server process.:

```
import time
from pcaspy import SimpleServer
from pcaspy.tools import ServerThread
server = SimpleServer()
server.setDebugLevel(4)
server_thread = ServerThread(server)
server_thread.start()
time.sleep(4)
server_thread.stop()
```

Qt GUI integration

qtqui.py shows how to combine it with Qt GUI event loop.

Example 5: Access Security Control

We already could refuse user written values in `write` method as done in Example 3. In addition it is possible to use access security rules as in EPICS database.

Define the access security rule

Suppose PV `MTEST:LEVEL` indicates the liquid nitrogen level and PV `MTEST:FILL` is the commanded amount of liquid nitrogen to refill. We want to refuse user's filling request when level is above 5.:

```
# test.as
# Access security rules
ASG(fill) {
    INPA($(P)LEVEL)
    RULE(1, READ)
    RULE(1, WRITE){
        CALC("A<5")
    }
}
```

It defines a rule *fill*, which grants read access but limits write access to `$(P)LEVEL` below 5. Refer to EPICS Application Developer's Guide about details.

Use security rules

In the database, `MTEST:FILL` has field *asg* set to the defined access rule *fill*.:

```
prefix = 'MTEST:'
pvdb = {
    'LEVEL': {},
    'FILL' : {'asg'  : 'fill',},
}
```

Before PVs are created, the access rules defined in the first step must be activated,:

```
...
server = SimpleServer()
server.initAccessSecurityFile('test.as', P=prefix)
server.createPV(prefix, pvdb)
...
```

Test

```
$ caput MTEST:LEVEL 2
Old : MTEST:LEVEL                    0
New : MTEST:LEVEL                    2

$ caput MTEST:FILL 5
Old : MTEST:FILL                     0
```

```
New : MTEST:FILL                    5

$ caput MTEST:LEVEL 6
Old : MTEST:LEVEL                   2
New : MTEST:LEVEL                   6

$ caput MTEST:FILL 8
Old : MTEST:FILL                    5
New : MTEST:FILL                    5

$ cainfo MTEST:FILL
MTEST:FILL
    State:            connected
    Host:             slslc06.psi.ch:5064
    Access:           read, no write
    Native data type: DBF_DOUBLE
    Request type:     DBR_DOUBLE
    Element count:    1
```

Other Tips

Hold string having more than 40 characters

`string` type is limited to 40 characters (at least in EPICS 3.14). To overcome this limit, use `char` type:

```
'STATUS' : {
    'type': 'char',
    'count' : 300,
    'value' : 'some initial message. but it can become very long.'
}
```

Later in the driver application, it can be accessed just like string parameter, e.g.:

```
self.setParam('STATUS', 'an error is happened')
print self.getParam('STATUS')
```

Alarm status and severity

- For numerical type, the fields *lolo*, *low*, *high*, *hihi* determine the alarm status and severity:

```
'VOLTAGE' : {
    'hihi' : 10,
    'high' :  5,
    'low'  : -5,
    'lolo' :-10
}
```

- For enumerate type, the fields *states* determine the alarm status:

```
'STATUS' : {
    'type' : 'enum',
    'enums':  ['OK', 'ERROR'],
```

```
    'states': [Severity.NO_ALARM, Severity.MAJOR_ALARM]
}
```

- For string type, the alarm status and severity can be changed by *Driver.setParamStatus()*.

Check out the reference *Driver.setParam()* and *Driver.setParamStatus()*, and alarm severity example.

Dynamic properties

For enumerate type, the choices are specified by field *enums* at startup. If in case the choices should be changed at runtime, *Driver.setParamEnums()* can be used. For numerical type, the precision, units, limits can also be changed by *Driver.setParamInfo()*. Check out the dynamic enums example.

To see the effect, use the following script:

```
import time
from CaChannel import ca, CaChannel

def monitor_callback(epics_arg, user_arg):
    if epics_arg['type'] == ca.DBR_CTRL_DOUBLE:
        print('units:', epics_arg['pv_units'])
    elif epics_arg['type'] == ca.DBR_CTRL_ENUM:
        print('enums:', epics_arg['pv_statestrings'])

enum = CaChannel('MTEST:ENUM')
enum.searchw()
enum.add_masked_array_event(ca.DBR_CTRL_ENUM, None, ca.DBE_PROPERTY, monitor_callback)

rand = CaChannel('MTEST:RAND')
rand.searchw()
rand.add_masked_array_event(ca.DBR_CTRL_DOUBLE, None, ca.DBE_PROPERTY, monitor_
↪callback)
rand.flush_io()

while True:
    time.sleep(1)
```

Now try to change the enum state and of MTEST:ENUM and units of MTEST:RAND.

```
$ caput MTEST:CHANGE 4
$ caput MTEST:RAND.EGU 'eV'
```

The script shall have the following output.

```
enums: ('ZERO', 'ONE')
units:
enums: ('ZERO', 'ONE', 'TWO', 'THREE')
units: eV
```

Create PVs using different prefix

Suppose one would want to create PVs with different prefix, maybe to distinguish their subsystem. It turns out to be quite easy, call *SimpleServer.createPV()* for each of them.:

```
prefix1='MTEST-1'
pvdb1={
'SIGNAL1': {},
}
prefix2='MTEST-2'
pvdb2={
'SIGNAL2': {},
}
...
    server.createPV(prefix1, pvdb1)
    server.createPV(prefix2, pvdb2)
```

Note however that the PV base name must not be the same, because *Driver* class uses PV base name as its identity.

Reference

SimpleServer

class pcaspy.**SimpleServer**

This class encapsulates transactions performed by channel access server. It stands between the channel access client and the driver object. It answers the basic channel access discover requests and forwards the read/write requests to driver object.

It derives from *caServer*. In addition to implement the virtual methods, it adds method *createPV()* to create the PVs and *process()* to process server requests.

```
server = SimpleServer()
server.createPV(prefix, pvdb)
while True:
    server.process(0.1)
```

createPV(*prefix*, *pvdb*)

Create PV based on prefix and database definition pvdb

Parameters

- **prefix** (*str*) – Name prefixing the *base_name* defined in *pvdb*
- **pvdb** (*dict*) – PV database configuration

pvdb is a Python *dict* assuming the following format,

```
pvdb = {
  'base_name' : {
    'field_name' : value,
  },
}
```

The base_name is unique and will be prefixed to create PV full name. This PV configuration is expressed again in a dict. The *field_name* is used to configure the PV properties.

Table 2.1: Database Field Definition

Field	Default	Description
type	'float'	PV data type. enum, string, char, float or int
count	1	Number of elements
enums	[]	String representations of the enumerate states
states	[]	Severity values of the enumerate states. Any of the following, Severity.NO_ALARM, Severity.MINOR_ALARM, Severity.MAJOR_ALARM, Severity.INVALID_ALARM
prec	0	Data precision
unit	''	Physical meaning of data
lolim	0	Data low limit for graphics display
hilim	0	Data high limit for graphics display
low	0	Data low limit for alarm
high	0	Data high limit for alarm
lolo	0	Data low low limit for alarm
hihi	0	Data high high limit for alarm
scan	0	Scan period in second. 0 means passive
asyn	False	Process finishes asynchronously if True
asg	''	Access security group name
value	0 or ''	Data initial value

The data type supported has been greatly reduced from C++ PCAS to match Python native types. Numeric types are 'float' and 'int', corresponding to DBF_DOUBLE and DBF_LONG of EPICS IOC. The display limits are defined by *lolim* abd *hilim*. The alarm limits are defined by *low*, *high*, *lolo*, *hihi*.

Fixed width string, 40 characters as of EPICS 3.14, is of type 'string'. Long string is supported using 'char' type and specify the *count* field large enough. 'enum' type defines a list of choices by *enums* field, and optional associated severity by *states*.

asyn if set to be True. Any channel access client write with callback option, i.e. calling *ca_put_callback*, will be noticed only when `Driver.callbackPV()` being called.

initAccessSecurityFile(*asfile*, *macro*)

Load access security configuration file

Parameters

- **filename** (*str*) – Name of the access security configuration file
- **subst** – Substitute macros specified by keyword arguments

---

Note: This must be called before `createPV()`.

---

process(*time*)

Process server transactions.

Parameters **time** (*float*) – Processing time in second

This method should be called so frequent so that the incoming channel access requests are answered in time. Normally called in the loop:

```
server = SimpleServer()
...
while True:
    server.process(0.1)
```

Driver

class pcaspy.**Driver**

This class reacts to PV's read/write requests. The default behavior is to accept any value of a write request and return it to a read request, an echo alike.

To specify the behavior, override methods *read()* and *write()* in a derived class.

__init__()

Initialize parameters database. This method must be called by subclasses in the first place.

read(*reason*)

Read PV current value

Parameters **reason** (*str*) – PV base name

Returns PV current value

This method is invoked by server library when clients issue read access to a PV. By default it returns the value stored in the parameter library by calling *getParam()*.

The derived class might leave this method untouched and update the PV values from a separate polling thread. See *Example 2: Interface to any shell command*, *Example 3: A Simulated Oscilloscope*.

Note: This method is called by the server library main thread. Time consuming tasks should not be performed here. It is suggested to work in an auxiliary thread.

---

write(*reason*, *value*)

Write PV new value

Parameters

- **reason** (*str*) – PV base name
- **value** – PV new value

Returns True if the new value is accepted, False if rejected.

This method is invoked by server library when clients write to a PV. By default it stores the value in the parameter library by calling *setParam()*.

Note: This method is called by the server library main thread. Time consuming tasks should not be performed here. It is suggested to work in an auxiliary thread.

---

getParam(*reason*)

retrieve PV value

Parameters **reason** (*str*) – PV base name

Returns PV current value

setParam(*reason*)

set PV value and request update

Parameters

- **reason** (*str*) – PV base name

- **value** – PV new value

Store the PV's new value if it is indeed different from the old. For list and numpy array, a copy will be made. This new value will be pushed to registered client the next time when *updatePVs()* is called. The timestamp will be updated to the current time anyway.

Alarm and severity status are updated as well. For numeric type, the alarm/severity is determined as the following:

value	alarm	severity
value < *lolo*	LOLO_ALARM	MAJOR_ALARM
lolo < value < *low*	LOW_ALARM	MINOR_ALARM
low < value < *high*	NO_ALARM	NO_ALARM
high < value < *hihi*	HIGH_ALARM	MINOR_ALARM
value > *hihi*	HIHI_ALARM	MAJOR_ALARM

For enumerate type, the alarm severity is defined by field *states*. And if severity is other than NO_ALARM, the alarm status is STATE_ALARM.

setParamStatus(*reason*, *alarm*, *severity*)
set PV status and severity and request update

Parameters

- **reason** (*str*) – PV base name
- **alarm** – alarm state
- **severity** – severity state

The PVs' alarm status and severity are automatically set in *setParam()*. If the status and severity need to be set explicitly to override the defaults, *setParamStatus()* must be called *after* *setParam()*.

setParamEnums(*reason*, *enums*, *states=None*)
set PV enumerate strings and severity states

Parameters

- **reason** (*str*) – PV base name
- **enums** (*list*) – string representation of the enumerate states
- **states** (*list*) – alarm severity of the enumerate states.

The number of elements in *states* must match that of *enums*. If *None* is given, the list is populated with *Severity.NO_ALARM*.

Note: The monitoring client will not receive this update. An explicit get is needed.

---

setParamInfo(*reason*, *info*)
set PV meta info, limits, precision, limits, units.

Parameters

- **reason** (*str*) – PV base name
- **info** (*dict*) – information dictionary, same as used in *SimpleServer.createPV()*.

callbackPV(*reason*)
Inform asynchronous write completion

Parameters **reason** (*str*) – PV base name

updatePVs()
Post update event on changed values

SimplePV

class pcaspy.**SimplePV**(*name*, *info*)
This class represent the PV entity and its associated attributes.

It is to be created by server application on startup. It derives from *PV* and implements the virtual methods.

Note: This is considered an internal class and should not be referenced by module users.

ServerThread

class pcaspy.tools.**ServerThread**(*server*)
A helper class to run server in a thread.

The following snippet runs the server for 4 seconds and quit:

```
server = SimpleServer()
server_thread = ServerThread(server)
server_thread.start()
time.sleep(4)
server_thread.stop()
```

__init__(*server*)

Parameters **server** – *pcaspy.SimpleServer* object

start()
Start the thread's activity.

It must be called at most once per thread object. It arranges for the object's run() method to be invoked in a separate thread of control.

This method will raise a RuntimeError if called more than once on the same thread object.

stop()
Stop the server processing

Development

The PCAS library is part of EPICS base. In version 3.14 it is found at *src/gdd* and *src/cas*. In version 3.15 it is found at *src/ca/legacy/gdd* and *src/ca/legacy/cas*. The library implements the channel access protocol. A server application could respond to the CA requests, when it implements the following interfaces.

caServer

class caServer

The virtual methods are,

pvExistReturn **pvExistTest**(**const** casCtx &*ctx*, **const** caNetAddr &*clientAddress*, **const** char **pPVAliasName*)
This function is called by the server library when it needs to determine if a named PV exists (or could be created) in the server application. This method should return *pverExistsHere* if server has this PV or *pverDoesNotExistHere* otherwise.

pvAttachReturn **pvAttach**(**const** casCtx &*ctx*, **const** char **pPVAliasName*)
This function is called **every** time that a client attaches to the PV. It should return a *PV* pointer on success, S_casApp_pvNotFound if this PV does not exist here.

The Python class *SimpleServer* implements this interface.

casPV

This is the basic entity representing a piece of data, with associated information like units, limits, alarm, timestamp etc. The C++ class *casPV* abstracts the interface to access the PV value. *PV* derives from *casPV* and further defines the interface to access the PV associated information. In addition it adds helper functions for asynchronous write.

class PV

Virtual methods

const char ***getName**()
Return the canonical (full) name for the PV.

unsigned **maxDimension**()

aitIndex **maxBound**(unsigned *dimension*)

Type	maxDimension	maxBound
Scalar	0	1
1D Array	1	number of elements in array

caStatus **getValue**(gdd &*value*)
The PV value.

caStatus **getPrecision**(gdd &*value*)
The PV precision.

caStatus **getUnits**(gdd &*value*)
The PV units.

caStatus **getEnums**(gdd &*value*)
The PV enumerated states.

caStatus **getLowLimit**(gdd &*value*)

caStatus **getHighLimit**(gdd &*value*)
The PV display/control limit

caStatus **getLowAlarmLimit**(gdd &*value*)

caStatus **getHighAlarmLimit**(gdd &*value*)
The PV alarm limit

caStatus **getLowWarnLimit**(gdd &*value*)

caStatus **getHighWarnLimit**(gdd &*value*)
The PV warning limit

caStatus **write**(**const** casCtx &*ctx*, **const** gdd &*value*)

caStatus **writeNotify**(**const** casCtx &*ctx*, **const** gdd &*value*)
The write interface is called when the server receives ca_put request and the writeNotify interface is called when the server receives ca_put_callback request.

A writeNotify request is considered complete and therefore ready for asynchronous completion notification when any action that it initiates, and any cascaded actions, complete.

Return S_casApp_postponeAsyncIO if too many simultaneous asynchronous IO operations are pending against the PV. The server library will retry the request whenever an asynchronous IO operation (read or write) completes against the PV.

caStatus **interestRegister**()
Called by the server library each time that it wishes to subscribe for PV change notification from the server tool via *postEvent()*.

caStatus **interestDelete**()
Called by the server library each time that it wishes to remove its subscription for PV value change events.

Helper methods

caStatus **postEvent**(**const** gdd &*event*)
Server application calls this function to post a PV *DBE_VALUE* | *DBE_LOG* event.

void **startAsyncWrite**()
Server application calls this function to initiate asynchronous write operation. This must be matched by a call to *endAsyncWrite()*.

void **endAsyncWrite**()
Server application calls this function to end asynchronous write operation.

bool **hasAsyncWrite**()
Return true if one asynchronous write is in progress.

bool **setAccessSecurityGroup**(**const** char **asgName*)
Server application calls this function to set the access security group name.

The Python class *SimplePV* implements this interface.

casChannel

This class could be used to finely control read/write access based on the client or other conditions. In C++ class *Channel*, the control access is implemented using Access Security Group. Library user does not need to instantiate this class, it is done inside *PV::createChannel*. As such this class is not exposed to Python.

gdd

class pcaspy.cas.**gdd**

gdd stands for General Data Descriptor. It is a generic, descriptive data container. Although designed to be generic, its usage in EPICS is limited to Portable Channel Access Server programming, more specifically in the getters of *pcaspy.SimplePV*.

classmethod gdd.**get**()
Retrieve the data. The gdd primitive types are up cast to Python types.

gdd	Python
aitEnumString aitEnumFixedString	str
aitEnumFloat32 aitEnumFloat64	float
aitEnumInt8 aitEnumUint8	str
aitEnumInt16 aitEnumUint16 aitEnumEnum16 aitEnumInt32 aitEnumUint32	int

Note: aitEnumInt8 and aitEnumUint8 are used to store char arrays.

classmethod `gdd.`**`put`**`(`*value*`)`
Store the data. The conversion table.

Input		gdd			
		Scalar		Atomic	
		numeric	string	numeric	string
gdd		copy dimension/bound info, then putDD			
numeric		putConvertNumeric | putNumericArray(size=1)			
string		putConvertString		convert to char array then putCharArray	
numpy	scalar	putConvertNumeric			
	array	putXXXDataBuffer			
sequence		setup gdd dimension/bound, then put(F)StringArray/putNumericArray			

classmethod `gdd.`**`setPrimType`**`(`*type*`)`
Force the GDD to change the primitive type of the data it describes. Changing the primitive type code is generally an unnatural thing to do. Force a GDD to change the application type, which effectively changes the high-level meaning of the data held within the GDD.

classmethod `gdd.`**`setStatSevr`**`(`*status*, *severity*`)`
Manipulate the status field of a GDD as a combination status and severity field.

classmethod `gdd.`**`setTimeStamp`**`()`
Manipulate the time stamp field of the GDD to the current time.

Release Notes

0.6.5 (09-05-2017)

- Fixed the anaconda upload on Travis Linux.
- Rebuilt PyPI/Anaconda packages to fix the wrong number of elements transfer when `ca_array_get_backack` requests with `0` count. (Issue #43)

0.6.4 (26-04-2017)

- Fixed the crash on Windows when epics base libraies are compiled using different visual studio version than python. (Issue #41)
- Fixed the sphinx docs build. (Issue #37)
- Added a spec file for rpmbuild.

0.6.3 (28-02-2017)

- Fixed that string type is wrongly converted to number. (Issue #24)

- Fixed that gdd.put crashes when input is string but primitiveType is other that Int8 or Uint8. (Issue #26)
- Added support of numpy array in gdd.put (Issue #28)
- Changed Python 3 support from 2to3 conversion to direct compatible source code. (Issue #30)
- Binary packages on PyPI is built with EPICS base 3.14.12.6. (Issue #25 and #29)
- Continuous integration/deployment configured at Travis and AppVeyor.

0.6.2 (02-08-2016)

- Fixed that the alarm/severity of string type PVs remain UDF/INVALID. (Issue #23)
- Fixed that the monitor event of string type PVs are wrongly posted as double. (Issue #24)

0.6.1 (01-07-2016)

- Fixed that PV graphics/control meta properties are not posted. See https://bugs.launchpad.net/epics-base/+bug/1510955 for the failure.
- Added support of EPICS 3.15
- Added `Driver.setParamInfo()` to set PV graphics/control meta properties, e.g. limits, units, precision.

0.6.0 (06-06-2016)

- Added support of request type DBR_CLASS_NAME in casPV base class. It is not overloaded in `SimplePV`. The default implementation returns an empty string.
- Warns and truncates if enums type PV has more than 16 states or any state string length > 25. (Issue #18)
- Fixed that PV of char type cannot be set to empty string. (Issue #14)
- Fixed that DBE_ALARM event was not posted. (Issue #13)

0.5.1 (10-10-2014)

- Fixed that alarm and warn limits are taken from *lolo*/*hihi* and *low*/*high* fields. (Issue #11).
- Fixed *example/alarm_severity.py* so that *MTEST:STATUS* and *MTEST:RAND* are writable.

0.5.0 (06-10-2014)

- Fixed that cas.py is not installed (Issue #10).
- Fixed that put callback may fail if driver invokes `Driver.callbackPV()` too soon(Issue #9). Thanks to Kay Kasemir.
- Added 16bit short integer type.
- Added printout of exceptions that are raised inside Python code.
- Added `Driver.setParamEnums()` to change the states of enumerate PV at runtime (Issue #4).
- Added basic logging support. To enable console logging:

```
logging.basicConfig(level=logging.DEBUG, format='%(asctime)s - %(levelname)s -
↪%(name)s: %(message)s')
```

- Changed so that timestamp is updated whenever `Driver.setParam()` is called. It was only updated when the values are new. The new behavior is the same as EPICS IOC.
- **Packaging changes:**
  - Binary packages are distributed through PyPI.
  - Document is hosted at Read the Docs.

0.4.1 (23-04-2013)

- Fixed PV initial status. By startup, it is UDF/INVALID.
- `Driver.setParam()` makes a copy of list/numpy.ndarray objects. This solves the racing condition, in which the value could be in the middle of updating while CA client reads the value. Thanks to Kay Kasemir.

0.4 (14-01-2013)

- Change from GPL to New BSD License for broader audience.
- Added __version__ info
- **Added access security control** access security control file can be loaded using `SimpleServer.initAccessSecurityFile()`. The loaded rules can then be assigned to PV's *asg* field. *example/access_control.py* provides an example and better explained in UserDocuments https://code.google.com/p/pcaspy/wiki/UserDocuments#Example_5:_Access_Security_Control
- **Added new type char.** It is used to represent a long string (>40 chars). And it behaves just like string parameters in driver. example/pysh.py provides a concrete example, in PV *COMMAND* and *OUTPUT*.
- **Added timestamp info.** The timestamp is set at each setParam call with new value different to current value.
- **Added alarm/severity info.** For *enum* type PV, the severity is configured by its *states* field. It is a list of severity states, which can be NO_ALARM, MINOR_ALARM, MAJOR_ALARM, INVALID_ALARM. If current state's severity is other than NO_ALARM, alarm is STATE_ALARM. For *int* or *float* type PV, the alarm state is configured by its *low*,*high*,*lolo*,*hihi* fields, in analogue to EPICS database. *example/alarm_severity* provides an example.

0.3 (21-09-2011)

- Fixed gdd vector memory leak introduced in 0.2
- Added casPV's writeNotify method for EPICS base 3.14.11+
- Release GIL for each C function call
- Added tools.ServerThread running server process in separate thread
- Added preliminary Qt GUI integration example using tools.ServerThread

0.2 (16-08-2011)

- Added Python 3 support
- Added numpy data types support
- Fixed the driver registration issue.
- Rework gdd put/get methods
- Added gdd unittest cases
- Remove Makefile in favor of setup.py

0.1 (19-07-2011)

- Initial release.

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Python Module Index

p

pcaspy, 16
pcaspy.tools, 20

Index

Symbols

__init__() (pcaspy.Driver method), 18
__init__() (pcaspy.tools.ServerThread method), 20

C

callbackPV() (pcaspy.Driver method), 19
caServer (C++ class), 20
createPV() (pcaspy.SimpleServer method), 16

D

Driver (class in pcaspy), 18

E

endAsyncWrite (C++ function), 22

G

gdd (class in pcaspy.cas), 22
get() (pcaspy.cas.gdd class method), 22
getEnums (C++ function), 21
getHighAlarmLimit (C++ function), 21
getHighLimit (C++ function), 21
getHighWarnLimit (C++ function), 21
getLowAlarmLimit (C++ function), 21
getLowLimit (C++ function), 21
getLowWarnLimit (C++ function), 21
getName (C++ function), 21
getParam() (pcaspy.Driver method), 18
getPrecision (C++ function), 21
getUnits (C++ function), 21
getValue (C++ function), 21

H

hasAsyncWrite (C++ function), 22

I

initAccessSecurityFile() (pcaspy.SimpleServer method), 17
interestDelete (C++ function), 22
interestRegister (C++ function), 22

M

maxBound (C++ function), 21
maxDimension (C++ function), 21

P

pcaspy (module), 16
pcaspy.tools (module), 20
postEvent (C++ function), 22
process() (pcaspy.SimpleServer method), 17
put() (pcaspy.cas.gdd class method), 23
PV (C++ class), 21
pvAttach (C++ function), 21
pvExistTest (C++ function), 21

R

read() (pcaspy.Driver method), 18

S

ServerThread (class in pcaspy.tools), 20
setAccessSecurityGroup (C++ function), 22
setParam() (pcaspy.Driver method), 18
setParamEnums() (pcaspy.Driver method), 19
setParamInfo() (pcaspy.Driver method), 19
setParamStatus() (pcaspy.Driver method), 19
setPrimType() (pcaspy.cas.gdd class method), 23
setStatSevr() (pcaspy.cas.gdd class method), 23
setTimeStamp() (pcaspy.cas.gdd class method), 23
SimplePV (class in pcaspy), 20
SimpleServer (class in pcaspy), 16
start() (pcaspy.tools.ServerThread method), 20
startAsyncWrite (C++ function), 22
stop() (pcaspy.tools.ServerThread method), 20

U

updatePVs() (pcaspy.Driver method), 19

W

write (C++ function), 21

write() (pcaspy.Driver method), 18
writeNotify (C++ function), 21