

Learning Python

Getting results for beamlines and scientific programming

Using python: Interacting with EPICS

Outline of topics to be covered

1. EPICS & beamline controls
2. Python & EPICS
3. The ca_util package
4. Accessing PVs in ca_util
5. Dealing with EPICS dropouts
6. Tips on developing beamline code



The Advanced Photon Source is an Office of Science User Facility operated for the U.S. Department of Energy Office of Science by Argonne National Laboratory

Controlling beamline equipment

- The APS (BCDA) standard is to control beamline equipment via EPICS. EPICS serves to manage communication to beamline devices such as motors, scalers, MCAs, temperature controllers, robots,...
- To then run instruments, we use a variety of methods for controlling EPICS:
 - MEDM GUIs to communicate and view
 - Spec scripts
 - Python scripts...
- Python advantages:
 - Fully featured programming language
 - integrate GUI, graphics, database calls & perhaps high-level math



A bit about EPICS

- EPICS is a complex protocol but I will only talk about some simple aspects of it. From a programming perspective, EPICS can be considered to be an array of variables that can either be read or set. These variables provide access to the instrument controls. Each variable is provided with a tag called a **PV** (process variable) that identifies the EPICS controller (IOC == crate or brick) that manages that connection.
 - Communication with a PV is via a protocol called **channel access (CA)**. Here we will discuss a few basic elements within the CA protocol
- You can only communicate with PVs on IOCs that are local to your subnet or that are repeated by the PV gateway (normally, not a concern).
- Communication between the client computer and the IOC is via ethernet.
- Communication between the IOC and the device may be routed in many different ways: VME, ethernet, RS-232, IEEE-488,...



Python EPICS support packages

- There are several Python support packages for EPICS are in use at the APS. The most widely used are ca_util and PyEpics
- ca_util:
 - found on all XSD beamlines via APSshare
 - supported by BCDA
 - not likely to be updated past Python 2.5
- PyEpics (<http://cars9.uchicago.edu/software/python/pyepics3/>):
 - from Matt Newville (CARS), also see talk at
<https://confluence.aps.anl.gov/download/attachments/2523138/PyEpics.pdf?version=1&modificationDate=1305235539000>.
 - this is not yet supported by BCDA, but this seems likely to happen



Ways to interact with EPICS in python

- In this presentation I will cover communicating with EPICS using three very simple EPICS communication methods:
 - caget: reads a PV, returning a value or set of values
 - caputw: sends a value to a PV, waiting for the value to be processed
 - caput: sends a value to a PV, returns immediately (no wait)
- There are more sophisticated types of EPICS communication:
 - monitor: writes a message or calls a routine every time a PV changes
 - alarms: respond to an out-of-range PV
 - much more
- I have found that caget, caput and caputw was all that I needed to implement the 11-BM automation and is all that I will cover here



What package should you use?

If you will need more than caput, & caget[w], I recommend that you work in PyEpics right away; be prepared for the “joy” of being the “first on the block”

- If you can work with only caput, & caget[w, I would suggest you use the BCDA supported ca_util package.

Code written using simple calls to ca_util.caput, ca_util.caget & ca_util.caputw can be eventually be transferred over to use of PyEpics pretty easily (possibly by changing only the import ca_util line).



The Advanced Photon Source is an Office of Science User Facility operated for the U.S. Department of Energy Office of Science by Argonne National Laboratory

Accessing the ca_util package

- The ca_util package must be accessed from /APSshare/bin/python

```
s11bmwork% /APSshare/bin/python
Python 2.5.2 (r252:60911, Jan 28 2009, 15:33:22)
[GCC 4.1.2 20070626 (Red Hat 4.1.2-13)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import ca_util
>>>
```



Possible setup problems in accessing the ca_util package

- If you get an error like this on importing ca_util:

```
s11bmsrv1:~ toby$ /APSshare/bin/python
Python 2.5.2 (r252:60911, Jan 28 2009, 15:33:22)
[GCC 4.1.2 20070626 (Red Hat 4.1.2-13)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import ca_util
warning: Not importing directory 'new': missing __init__.py
>>>
```

- The problem is likely to be some out-of-date environment variables that interfere with ca_util in Python:

```
s11bmwork% env | grep -i python
PYTHONSTARTUP=/APSshare/epics/startup/2007_07_31/pythonConfig.py
PYTHONPATH=/APSshare/epics/extensions_2007_07_31/lang/python:/APSshare/
epics/extensions_2007_07_31/lib/linux-x86-fc6
```

- I see this on some accounts at 11-BM (probably due to settings in the .cshrc/.login file)



Work-around for ca_util package load problems

Here is how to invoke Python if you have the previous problem

- bash:

```
s11bmsrv1:~ toby$ export PYTHONPATH=
s11bmsrv1:~ toby$ export PYTHONSTARTUP=
s11bmsrv1:~ toby$ /APSshare/bin/python
Python 2.5.2 (r252:60911, Jan 28 2009, 15:33:22)
[GCC 4.1.2 20070626 (Red Hat 4.1.2-13)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import ca_util
>>>
```

- csh/tcsh

```
s11bmsrv1% unsetenv PYTHONPATH
s11bmsrv1% unsetenv PYTHONSTARTUP
s11bmsrv1% /APSshare/bin/python
Python 2.5.2 (r252:60911, Jan 28 2009, 15:33:22)
[GCC 4.1.2 20070626 (Red Hat 4.1.2-13)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import ca_util
>>>
```



Expanding the ca_util buffer size

- If you will transfer very long sets of values between Python and EPICS, then the default channel access buffer size may be too small. This can be increased by setting the `EPICS_CA_MAX_ARRAY_BYTES` environment variable to the number of bytes to be transferred (e.g. 8 times the number of doubles) plus a bit extra (~50 bytes).
 - example: on 11-BM the Struck scaler may return >100,000 bytes
- tcsh/csh:

```
setenv EPICS_CA_MAX_ARRAY_BYTES 1280008
```

- bash

```
export EPICS_CA_MAX_ARRAY_BYTES=1280008
```

This needs to be done before python is started. It can be done from inside python, but only before importing `ca_util`.



The Advanced Photon Source is an Office of Science User Facility operated for the U.S. Department of Energy Office of Science by Argonne National Laboratory

Reading a PV

- Reading a PV is easy: call `ca_util.caget(PV)` where PV is a string containing the name of the PV

```
>>> import ca_util
>>> ca_util.caget("11bmb:m28.RBV")
94.40000000000006
>>> ca_util.caget("11bmb:CS:AlarmCode.SVAL")
'End phase complete'
>>> a = ca_util.caget("11bmb:3820:mca12")
>>> len(a)
30000
```

- Note that the type that is returned depends on the PV



Setting a PV

- To set a PV one calls caput or caputw: `ca_util.caput(PV, val)` or `ca_util.caputw(PV, val)`
 - The difference is that `caput` returns immediately while `caputw` returns after the change has taken effect.
- Example: the code below moves a motor, but if `wait` is `True`, the python script pauses until the motor reaches the position (when `caputw` returns).

```
# Drive the Diffractometer 2-theta axis to a preset position
# wait = False: the motor starts moving but execution continues
def Drive11BM2theta(position,wait):
    if config.debug: print "Drive 11-BM to %f Two-theta" % position
    if wait:
        ca_util.caputw("11bmb:m28.VAL",position)
    else:
        ca_util.caput("11bmb:m28.VAL",position)
```



Warning: caput and caget do not always work

- The caput[w] channel access methods do not check that the message you are sending actually arrives to the intended interface. Also, both caput and caget can sometimes timeout or otherwise fail.
 - The access may not go through due to a network collision or problem
 - The EPICS controller may get the message, but the attached device may fail to respond
- When you write a script, you need to consider that a channel access command may fail:
 - It may generate an exception
 - It may run normally, but fail to do what you wanted
- For scripts where damage could occur if things are not done in the right sequence or that will run unattended (and should not fail at 2 am), write **code defensively** -- assuming that you may need to catch an exception and/or check that the action you intended actually happened.



Defensive coding

- Here is an example of some code I use for opening the filter-box shutter

```
# open the shutter: loop until it is actually open
for i in range(10):
    try:
        if (ca_util.caget('11bmb:xias:Status3') == 1 and
            ca_util.caget('11bmb:xias:Status4') != 1): return False
        ca_util.caputw("11bmb:xias:openShutter.PROC",1)
        gui.Sleep(1.5) # wait for update
    except:
        print "ignoring exception in Set11BMShutter open"
        gui.Sleep(0.1)
    gui.Log11BMmessage("Unable to open shutter after %d attempts" % i)
return True
```

- This code will ignore up to 10 exceptions before giving up
- It checks status bits to make sure the two blades are open
- If not, it sends an open command, waits and then tests again

Note that I tend to use a convention where routines return True in case of error so that I can code like this

```
if DoSomething(): ErrorLogger("message")
```



Waiting for something to happen

- Much of instrument control code is waiting for something to happen.
- One can use callbacks – these are routines that are called when an EPICS event happens. This can provide very fast response, but is potentially complex.
- Unless I need a response in less than 0.1 sec, my preference is to have a loop that checks a PV and then waits for some period, using the `time.sleep()` function.
 - GUI programming note: during a `time.sleep` delay, the wxPython event loop cannot respond, so one wants to call `wx.Yield()` to allow for screen updates and button events to occur, etc.
 - I use this sleep function in GUI code (`config.WaitTick` is 0.1 sec):

```
# perform a sleep command, but also do periodic screen updates
def Sleep(seconds):
    wait = 0
    while wait <= seconds:
        time.sleep(config.WaitTick)
        wait += config.WaitTick
        wx.Yield()
    return
```



Alignment programs

- I typically drive alignment scans point-by-point in python (rather than scan a fixed range) so that I can end the scan as soon as I am well past the peak or so that I don't stop early if I am close to the peak.
- Outline of my peak finding code:
 - Support routines:
moveScanLoc -- moves to next location;
appends location to X array
doCount – starts a counter, waits for it to finish & appends intensity to Y
checkPeak – looks at Y values and determines if a peak has been found; or if the scan range should be restarted earlier

There is a need for some generic multiaxis peak-finding routines that do sparse "hunts" and then reduce step sizes as they get close to the peak.

```
scanrange = ...
repeat = True
while repeat:
    X = []
    Y = []
    while True:
        if moveScanLoc(X, scanrange):
            print 'no peak in range'
            raise NoPeakError, "out of range"
        doCount(Y)
        stat = checkPeak(X, Y, scanrange)
        if stat == 'done':
            repeat = False
            break
        elif stat == "restart":
            scanrange = ...
            break
```



More on PV names

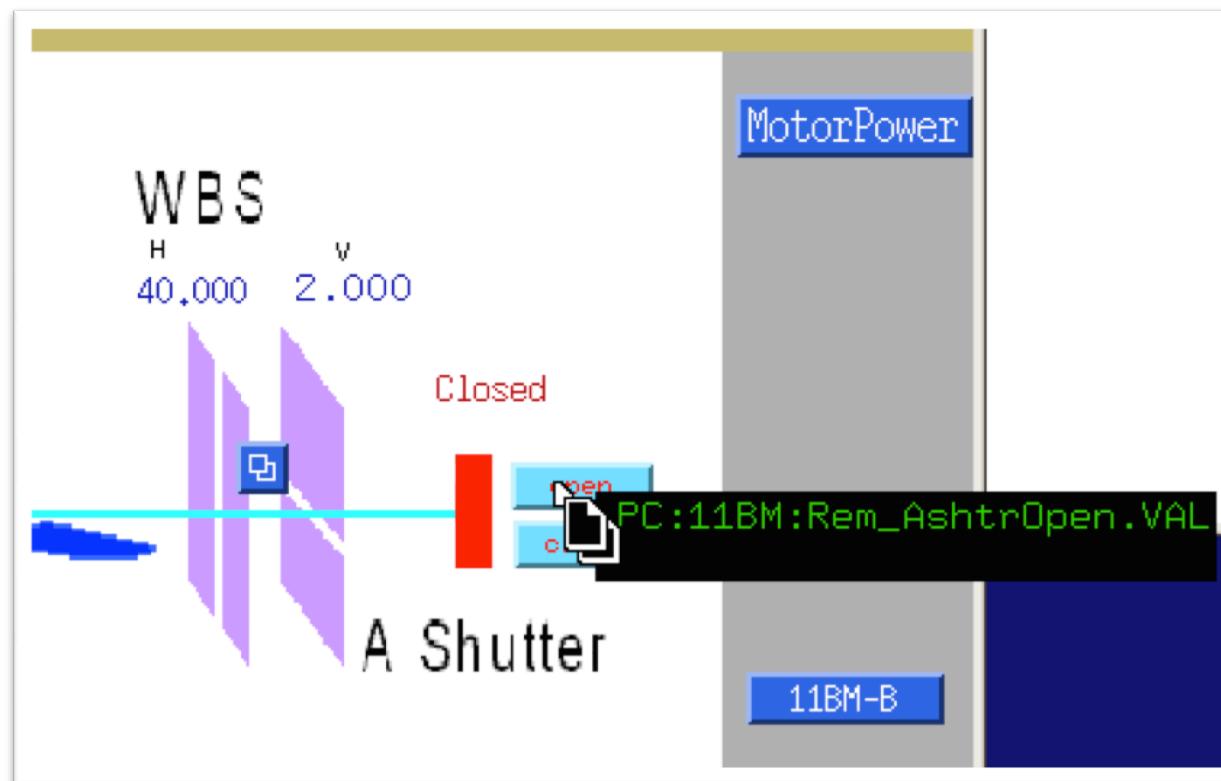
- So how do you know what PVs to use?
 - in most cases you should first perform the action you want to place in a script from the MEDM interface.
 - As you do this, keep notes on what you do



The Advanced Photon Source is an Office of Science User Facility operated for the U.S. Department of Energy Office of Science by Argonne National Laboratory

Getting PV names from MEDM

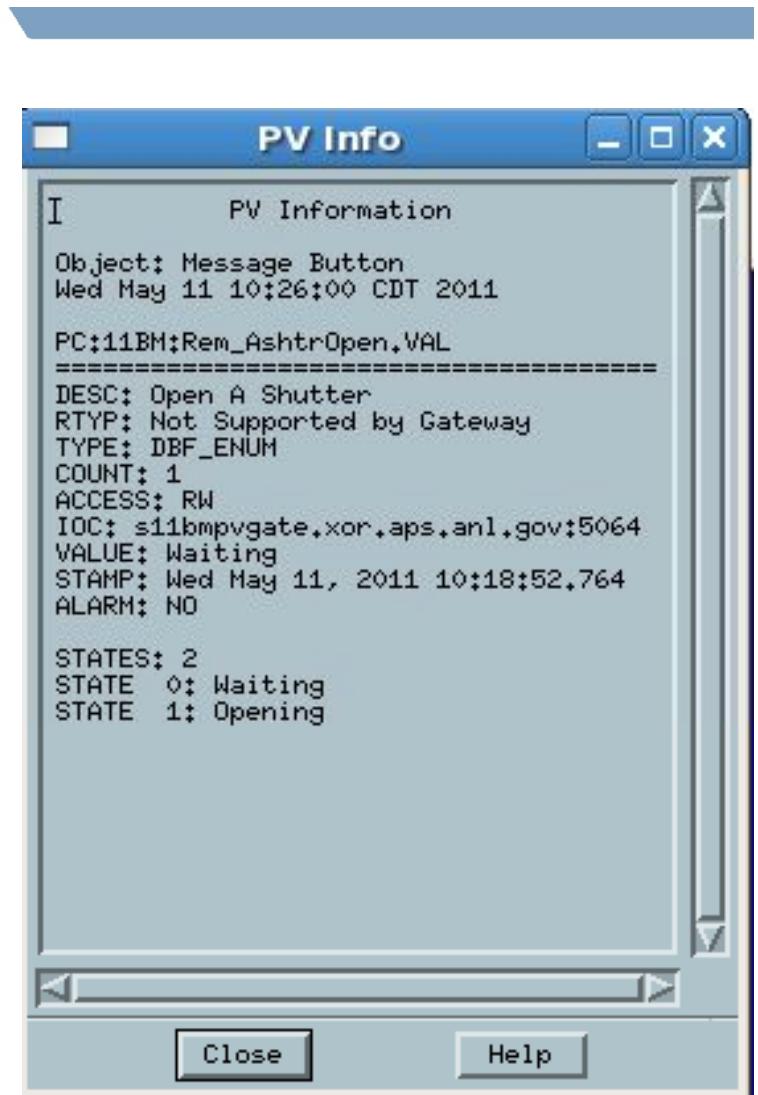
- If you press the middle mouse button on a button or value, MEDM will show the name of the associated widget
 - If you hold the button down and move to a X11 program, you can sometimes paste the PV name. (Alternately look for nameCapture)



The Advanced Photon Source is an Office of Science User Facility operated for the U.S. Department of Energy Office of Science by Argonne National Laboratory

Information about PVs

- a right mouse click in MEDM provides a menu



- Most useful is PV Info, which provides details on a PV

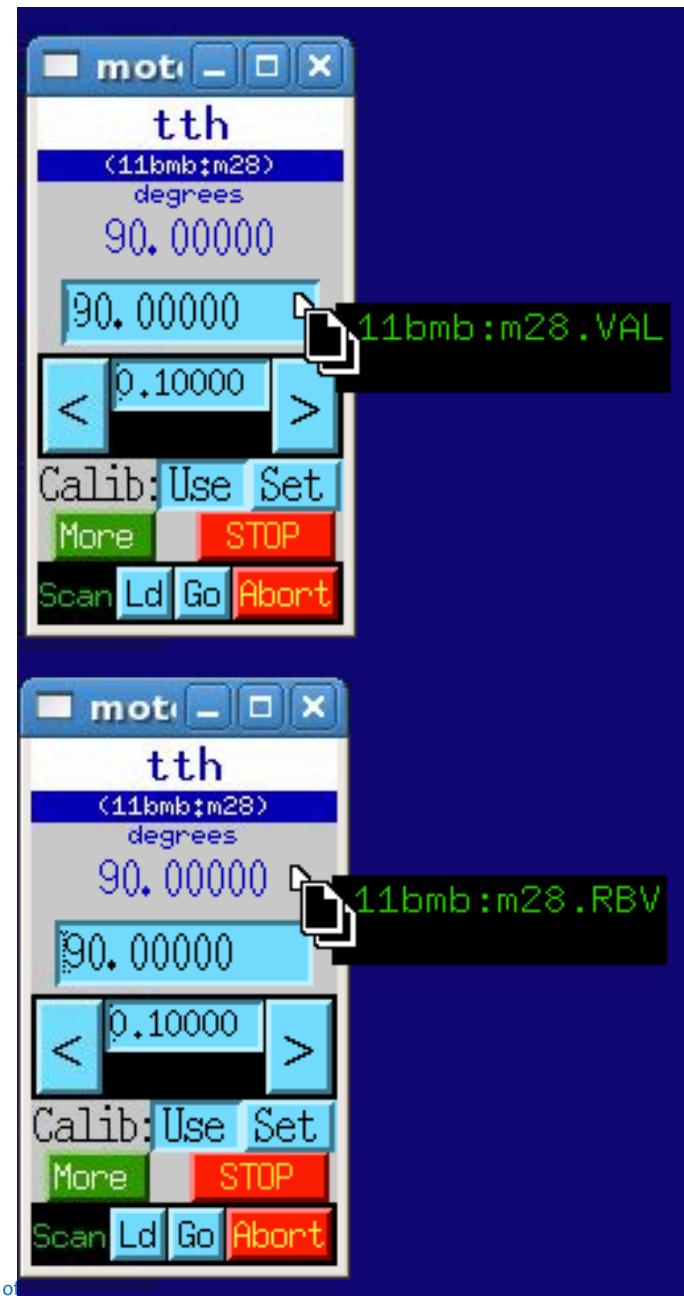


The Advanced Photon Source is an Office of Science User Facility operated for the U.S. Department of Energy Office of Science by Argonne National Laboratory

Programming motors

Note that motors have many associated PVs

- The base name for this motor is shown in blue (11bmb:m28)
- For motor position there are two important PVs:
<IOC>:<motor>.VAL Set (caput) this to specify where the motor should drive
<IOC>:<motor>.RBV Read (caget) this to see where the motor is actually located
 - Note that a change tells you that the motor driver received a message from the IOC, but not that the motor moved if it is stalled or turned off.
 - More shows many more PVs, for example motor limits



Now use the PVs

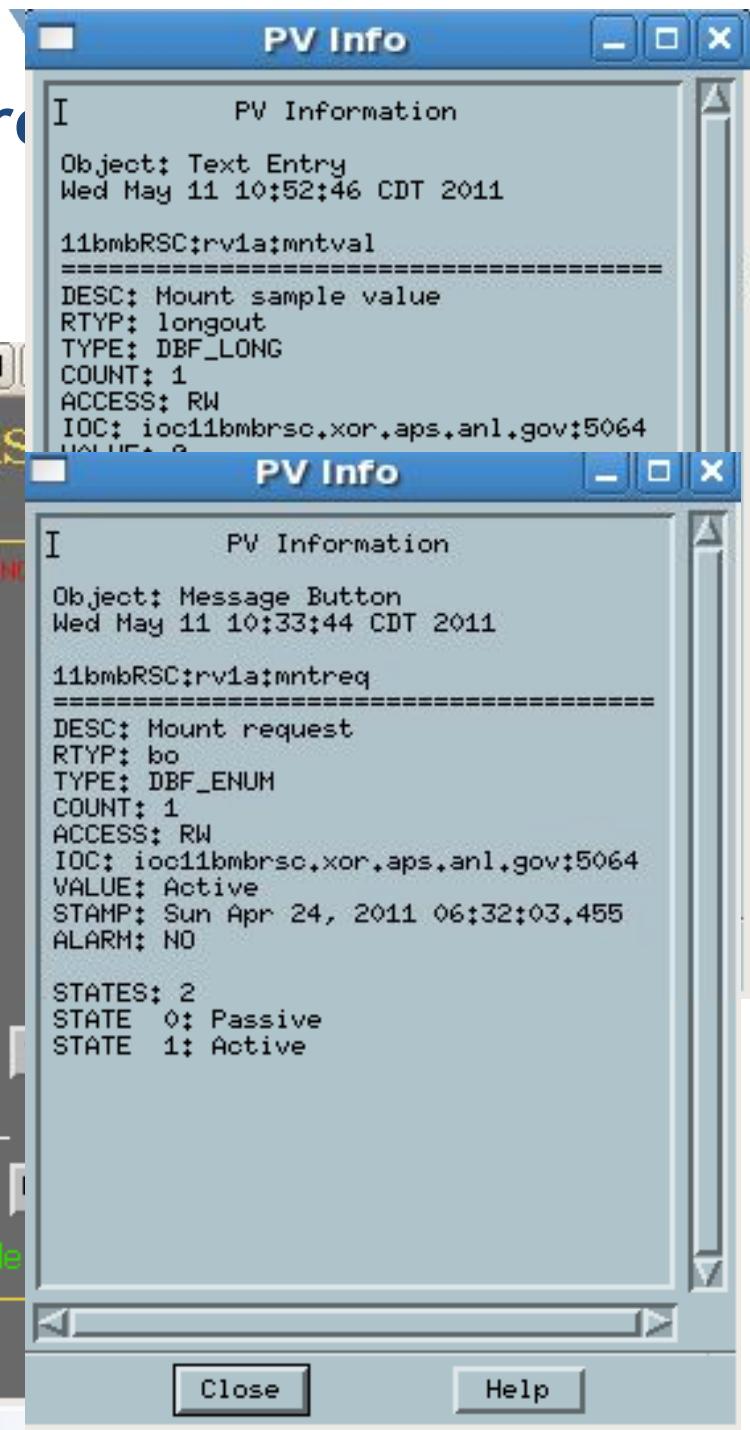
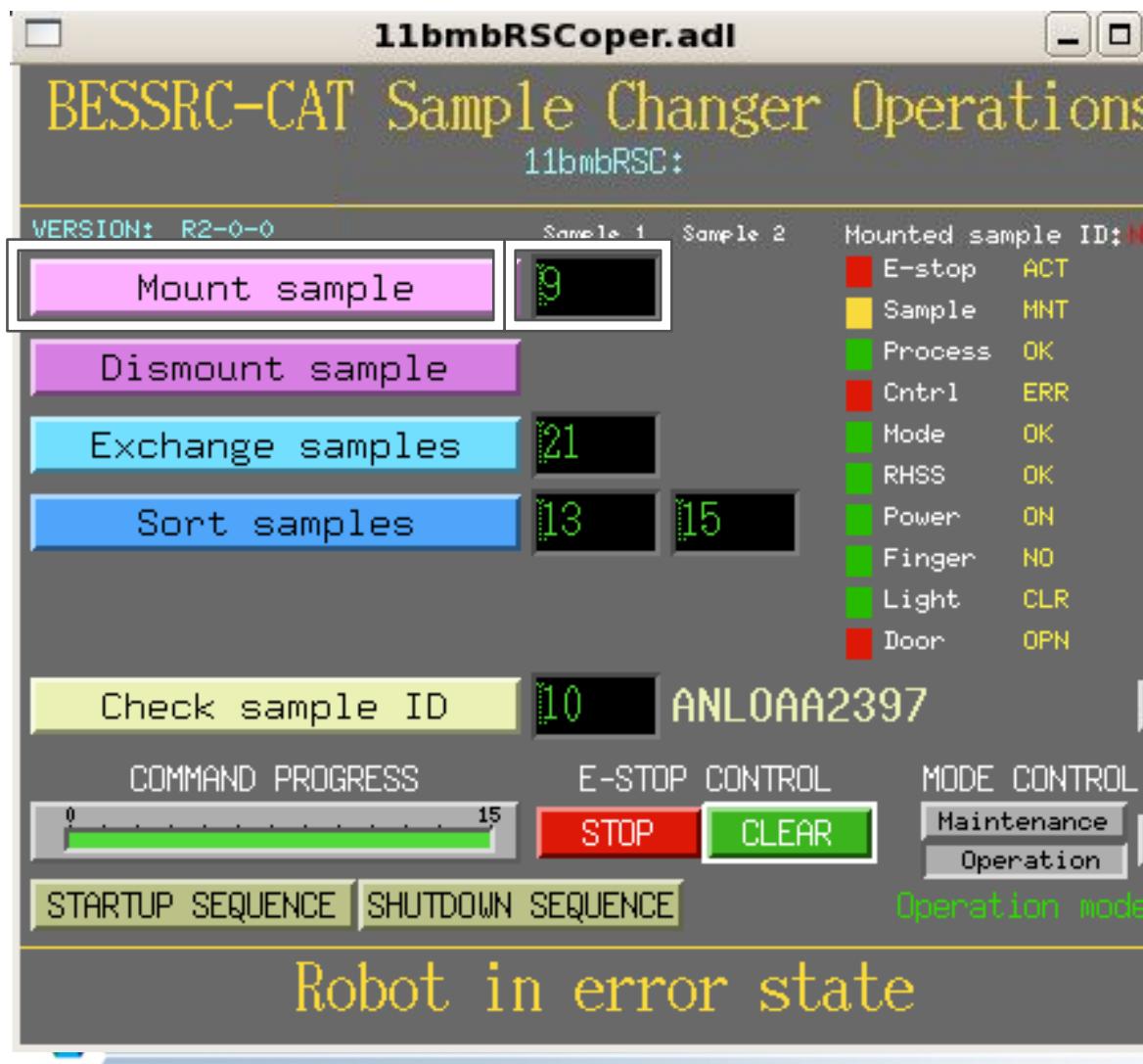
- Here are the routines I use to interface with the diffractometer 2theta motor. Note that I take my chances on reading the PV (with respect to getting the position) and that due to the way data collection is done on 11-BM, a skipped drive will only slow down the instrument, so I can safely ignore an exception.

```
# read the Diffractometer 2-theta position
def Get11BM2theta():
    return ca_util.caget("11bmb:m28.RBV")
```

```
# Drive the Diffractometer 2-theta axis to a preset position
# set wait to True if the routine should sleep until the position is reached
# wait = False: the motor starts moving but execution continues
def Drive11BM2theta(position,wait):
    try:
        if config.debug: print "Drive 11-BM to %f Two-theta" % position
        if wait:
            ca_util.caputw("11bmb:m28.VAL",position)
        else:
            ca_util.caput("11bmb:m28.VAL",position)
    except: print "ignore exception in Drive11BM2theta"
```



More complex systems may require interacting with many PVs



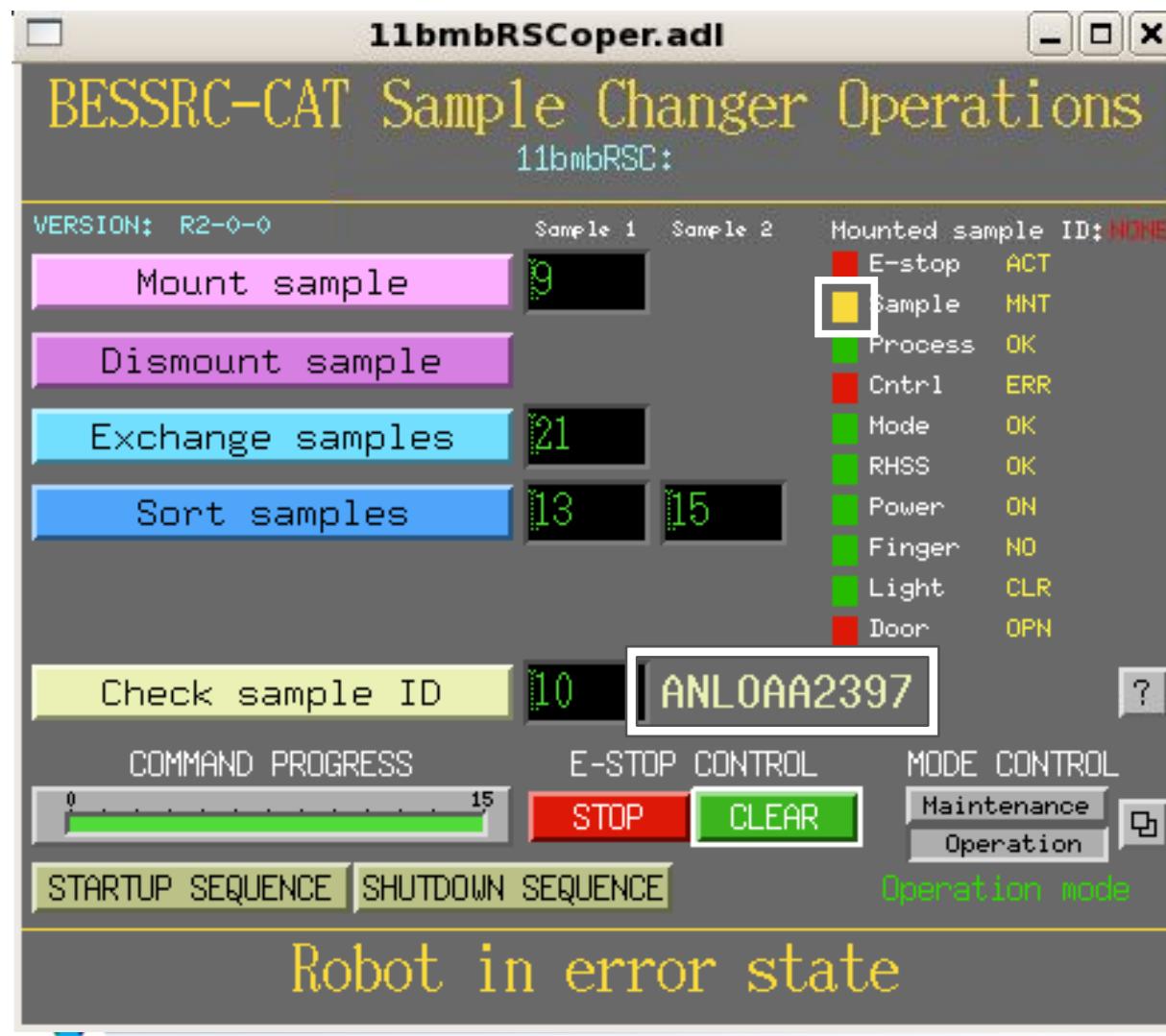
Programming the robot

- Sample code (best inside a try block that loops until success)

```
# tell robot the sample number to use
ca_util.caputw("11bmbRSC:rv1a:mntval",position)
# trigger loading the sample
ca_util.caputw("11bmbRSC:rv1a:mntreq",0)
ca_util.caputw("11bmbRSC:rv1a:mntreq",1)
```



Where possible, test other PVs to check that the action occurred.



PV Info

I PV Information

Object: Text Monitor
Wed May 11 10:49:40 CDT 2011

11bmbRSC:rv1at:barcode

=====

DESC: Bar code reading
RTYP: stringin
TYPE: DBF_STRING
COUNT: 1
ACCESS: RW
IOC: ioc11bmbrc,xor,aps,anl.gov:5064
VALUE: ANLOAA2397
STAMP: Sun Apr 24, 2011 06:32:10,399
ALARM: NO

PV Info

I PV Information

Object: Rectangle
Wed May 11 10:48:29 CDT 2011

11bmbRSC:omm01:reg06:bi07

=====

DESC: SAMPLE MOUNTED
RTYP: bi
TYPE: DBF_ENUM
COUNT: 1
ACCESS: RW
IOC: ioc11bmbrc,xor,aps,anl.gov:5064
VALUE: MNT
STAMP: Wed May 11, 2011 10:48:29,144
ALARM: NO

STATES: 2
STATE 0: MNT
STATE 1: NO

Close **Help**

Did a sample get loaded?

- This routine reads the status bit. The code can then check that a sample was actually loaded when I asked for that

```
# returns True if a sample is loaded, false otherwise
def CheckForLoaded11BMsample():
    i = 0
    while i < 10:
        i += 1
        try:
            if (ca_util.caget("11bmbRSC:omm01:reg06:bi07") == 1):
                return False
            else:
                return True
        except:
            print "exception in CheckForLoaded11BMsample"
            gui.Sleep(0.1)
    else:
        gui.Log11BMmessage("Abort: CheckForLoaded11BMsample exceptions")
```



Complex sequences of PVs require attention for error recovery

- The code to mount the sample (1st box) should not be in the same try block as the call to read the barcode (2nd box).
 - Why? because if the sample mount fails you want to repeat it, but if the barcode read fails, you don't want to repeat the sample mount command, if it already succeeded. (There should be no problem in repeating any of the commands in the 1st box).
 - N.B. my real code logs errors and aborts after a number of exceptions

```
while CheckForLoaded11BMsample():
    try:
        # tell robot the sample number to use
        ca_util.caputw("11bmbRSC:rv1a:mntval",position)
        # trigger loading the sample
        ca_util.caputw("11bmbRSC:rv1a:mntreq",0)
        ca_util.caputw("11bmbRSC:rv1a:mntreq",1)
    except:
        pass
```

```
while True:
    try:
        barcode = ca_util.caget("11bmbRSC:rv1a:barcode")
        break
    except:
        pass
```



Do what I say not...

- Note that in all my examples, I have hard-coded PV names. This is not a good thing. A better choice is to define PVs in one place using a dict (perhaps in a configuration file).

```
# define PVs
PV = { 'D2ThR': "11bmb:m28.RBV",
       'D2ThS': "11bmb:m28.VAL",
       ...
     }

# read the Diffractometer 2-theta position
def Get11BM2theta():
    return ca_util.caget(PV[ 'D2ThR' ])

def Drive11BM2theta(position,wait):
    try:
        if wait:
            ca_util.caputw(PV[ 'D2ThS' ],position)
        ...
    
```



Other tips

- When developing EPICS control code, I frequently work with several open windows:
 1. MEDM window(s)
 2. a xterm window where I drag PV names (I may also use caput and caget from unix shell) for tests
 3. a python shell where I try out commands
 4. an editor, where I copy commands that did what I want into a script
- Data collection codes really benefit from storage of “metadata” and event logging.
 - Consider hierarchical storage: NeXus (see <http://trac.mcs.anl.gov/projects/nexpy/>) or HDF5 (see <http://www.pytables.org> or <http://code.google.com/p/h5py/>).
 - Consider the built-in logging module (<http://docs.python.org/library/logging.html>).
 - Consider database use (MySQL, etc.) as a way to keep collection protocols and results.

