DIVERSE USES OF PYTHON ON DIAMOND

Michael Abbott, Tom Cobb, Ian Gillingham, Mark Heron

Diamond Light Source, Oxfordshire, UK.

Abstract

Diamond Control Systems Group has used Python for a range of control system applications. These include scripts to support use of the application build environment, client GUIs and integrated with EPICS as EPICS Channel Access servers and concentrators. This paper will present these applications and summarise our experience.

Channel Access Bindings

The catools library provides three functions for access to EPICS “process variables” over channel access:

`caget(pvs, …)`
Returns a single snapshot of the current value of each PV.

`caput(pvs, values, …)`
Writes values to one or more PVs.

`camonitor(pvs, callback, …)`
Receive notification each time any of the listed PVs changes.

```
# Library version specification required for dls libraries
from pkg_resources import require
require('cothread')

import cothread
from cothread.catools import *

# Using caput: write 1234 into PV1.  Raises exception
# on failure
caput('PV1', 1234)

# Print out the value reported by PV2.
print caget('PV2')

# Monitor PV3, printing out each update as it is received.
def callback(value):
    print 'callback', value
camonitor('PV3', callback)

# Now run the camonitor process until interrupted by Ctrl-
C.
cothread.WaitForQuit()
```

Working with Values

There are two types of values returned by catools functions: “augmented values” and “error codes”. The `caput` function only returns an error code value (which may indicate success), while `caget` and `camonitor` will normally return augmented values, but will return an error code on failure. (To be precise, `camonitor` delivers values to its callback function.)

The following fields are common to both types of value: `.ok` and `.name`. This means that is is always safe to test `value.ok` for a value returned by `caget` or `caput` or delivered by `camonitor`.

Augmented Values

Augmented values are normally Python or numpy values with extra fields: the `.ok` and `.name` fields are already mentioned above, and further extra fields will be present depending on format requested for the data. As pointed out above, `.ok` is only false for error returns.

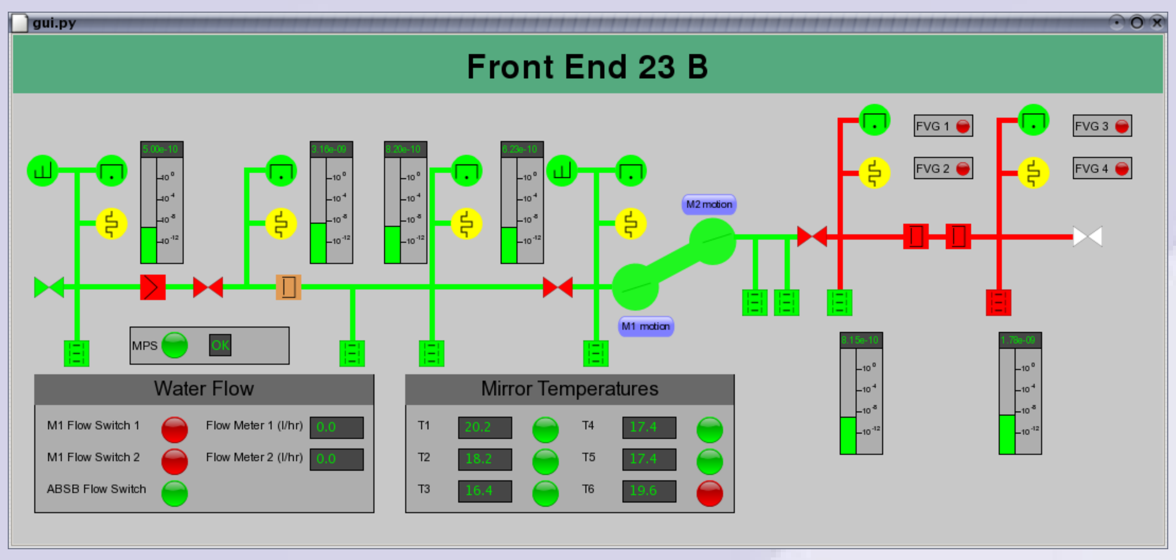
Four different types of augmented value are returned: strings, integers, floating point numbers or arrays, depending on the length of the data requested — an array is only used when the data length is >1.

In almost all circumstances an augmented value will behave exactly like a normal value, but there are a few cases where differences in behaviour are observed (these are mostly bugs). If this occurs the augmentation can be stripped from an augmented value value by writing `+value` — this returns the underlying value in all cases.

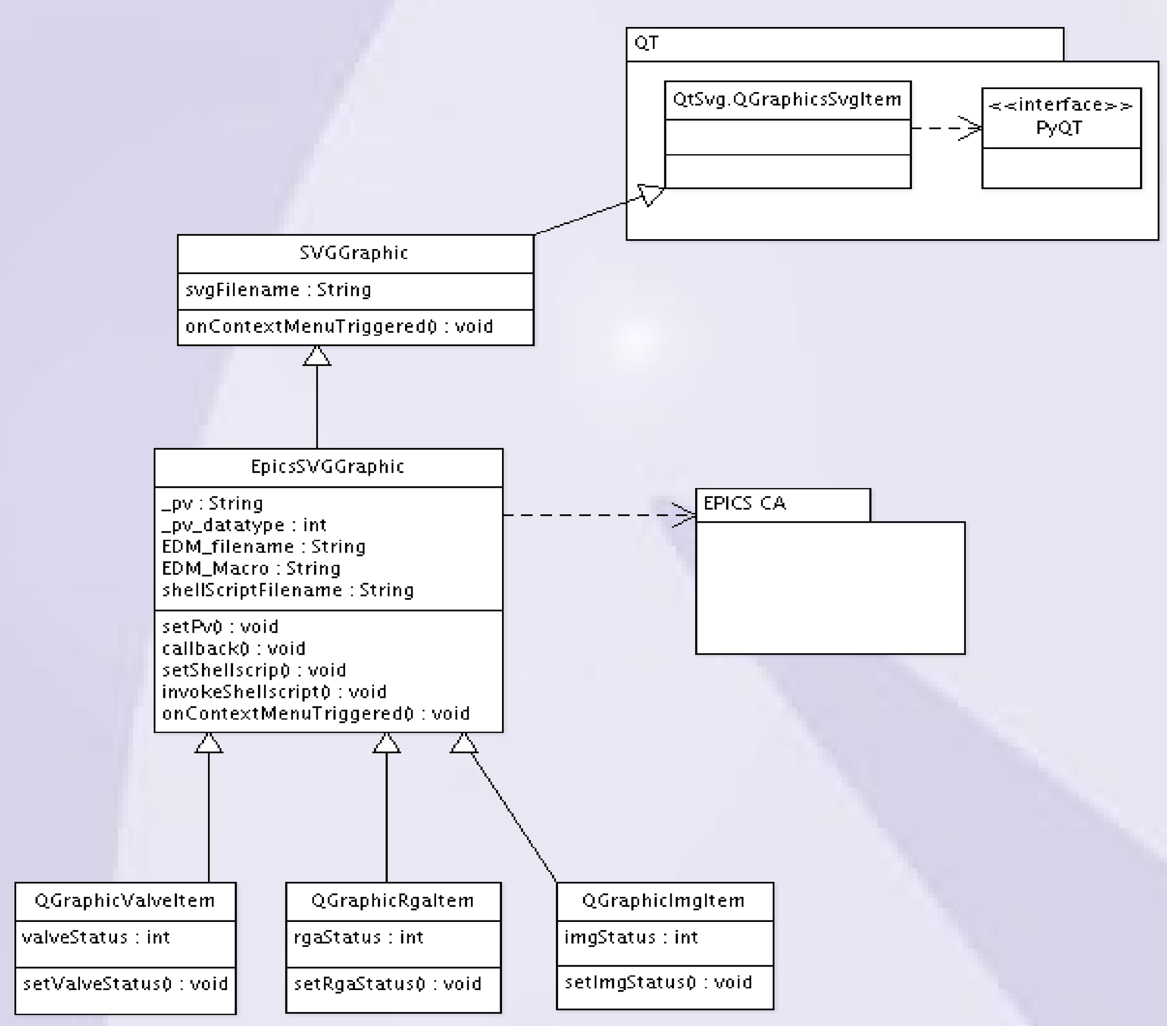
The type of augmented values is determined both by parameters passed to `caget` and `camonitor` and by the underlying datatype. Both of these functions share parameters `datatype`, `format` and `count` which can be used to force the type of the data returned.

Control System User Interface

A graphical user interface has been implemented at Diamond, based on the QT interface with Python (PyQt) and Channel Access bindings. The screenshot below shows an example of this applied to a photon beam front-end.



The following class diagram highlights the design pattern adopted to facilitate the implementation of widgets with specific characteristics. Only three widget classes are shown as examples in this diagram.



The EPICS Channel Access (CA) interface is via the Python Channel Access package (see Section 1). An example implementation (from Diamond's Front Ends interface) is abstracted in the EpicsSVGGraphic class. The application framework implements the modern QT scene/view framework. The view layout is specified by subclassing QtGui.QGraphicsView, instantiating all widgets in the constructor, defining their positions and setting their PV identifiers.
When EpicsSVGGraphic is instantiated, it subscribes to updates of the given EPICS PV, by supplying its callback function to the CA interface. Update processing, specific to a graphical widget, is realised simply by overriding the base-class callback. For instance, a valve widget will change the fill-colour of the graphical element, depending on the new valve status.
All widgets derived from EpicsSVGGraphic, also inherit full clipboard copy functionality (XDND protocol), tooltips and context menus.

Python in Simulations

EPICS based photon beamlines at Diamond are increasingly using Asyn as an interface layer between Device and Driver Support. This abstraction allows the low level driver to be replaced with a simulation without modifying the upper levels of the structure. These simulations support early testing, not only of high level applications including EDM panels and GDA for data acquisition, but also core modules such as Asyn and Stream Device.

Python modules have been written to simulate a device in two different ways. The first method is used for serial (RS232, RS485, simple TCP/IP) devices. The second is used for more complicated devices like cameras or scaler cards. By embedding python in the Linux IOC, both classes can be instantiated in the simulation startup script.

Simulated Device

The Python class serial_device wraps either a TCP server or a Linux pseudo serial port, deals with I/O and terminators, and provides scheduling functionality. The programmer is required to code a reply method suitable for the device. Below is the code for a device that has one internal value. It can be read by sending "?" and written by sending anything else.

```
class my_serial(serial_device):
    # set a terminator and internal val
    Terminator = "\r\n"
    val = 1

    def reply(self, command):
        # return reply to <command>
        if command=="?":
            return self.val
        else:
            self.val=command
            return "OK"
```

Simulated Driver Support

The Python class pyDrv registers itself as an Asyn port with a variety of interfaces, provides scheduling and callback functionality and handles type conversion to and from Python native types. The programmer is required to code suitable write and read methods. The code below is for a simple example that keeps an internal dictionary object of values, and allows access to these via a series of commands.

```
class my_asyn(pyDrv):
    # supported list of asyn commands
    commands = ["A","B","C","D"]
    # internal dictionary of values
    vals = {"A":1,"B":"BEE","C":3.4,"D":[1,2,3,4]}

    def write(self,command,signal,value):
        # write <value> to internal dict
        self.vals[command] = value

    def read(self,command,signal):
        # return value from internal dict
        return self.vals[command]
```