CaChannel Documentation

Release 3.1.3

Xiaoqiang Wang

Oct 01, 2020

CONTENTS

CHAPTER

ONE

OVERVIEW

CaChannel.CaChannel wraps the low level *ca* functions into a Python class. It provides methods to operate on the remote process variable over channel access connection.

It was developed in 2000 by Geoff Savage using caPython extension Later on the CA library becomes multi-threaded with EPICS base 3.14. The original developer did not address this change however.

In 2008 during the transition to EPICS 3.14, a new implementation of CaChannel interface, version 2.x, was developed based on PythonCA extension from Noboru Yamamoto. It was highly backwards compatible with the original implementation based on caPython.

In 2014, package caffi was created in place of caPython or PythonCA extension to expose the Channel Access API using cffi. It aimed to create a Pythonic API of one-to-one map to the counterpart C functions. After that has been accomplished, CaChannel interface was re-implemented using the `caffi.ca` module, versioned 3.x.

In 2015-16, with all previous experiences, the `CaChannel.ca` module was rewritten from scratch using Python/C API. The new `CaChannel.ca` module has a compatible API with `caffi.ca`. Because of that, the CaChannel interface can use `caffi.ca` by setting environment variable `CACHANNEL_BACKEND=caffi`. This is also the fallback backend if no EPICS installation exists or the C extension fails to import.

CHAPTER

TWO

CONTENTS

2.1 Installation

2.1.1 Binary Installers

Anaconda

Packages for Anaconda can be installed via:

```
conda install -c paulscherrerinstitute cachannel
```

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,

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

Egg

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

```
$ sudo easy_install cachannel
```

Or install only for the current user:

```
$ easy_install --user cachannel
```

2.1.2 Source

If a binary package is not available for your system, you can build from source. If you have a valid EPICS base installation, as described by *Getting EPICS*, the C extension will be compiled. Otherwise it will instead use caffi as the backend.

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/CaChannel/CaChannel.git
```

On Linux, the python header files are normally provided by package like *python-devel* or *python-dev*.

`numpy` is optional, but it can boost the performace when reading large waveform PVs, which are common for areaDetector images.

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
	64bit	windows-x64
macOS	PPC	darwin-ppcx86
	Intel	darwin-x86

- Run `make`.

Build

As soon as the epics base libraries are ready, it is simple,

- On Widnows:

```
> C:\Python27\python.exe setup.py install
```

- On Linux/macOS:

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

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
```

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

```
$ rpmbuild -ta CaChannel-3.0.0.tar.gz
```

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

2.2 Short Tutorials

2.2.1 Synchronous Actions

Each of the actions (search, put, get) ending with a "w" signify that the action completes before the function returns. In CA terms this means that a call to ca_pend_io() is issued to force the action to process and wait for the action to complete. When an exception occurs the offending CA status return is printed using print ca.message(status).

```
from CaChannel import CaChannel, CaChannelException
try:
    chan = CaChannel('catest')
    chan.searchw()
    chan.putw(12)
    chan.getw()
except CaChannelException as e:
    print(e)
```

2.2.2 Multiple Synchronous Actions

Connection

Multiple channel access connection requests.

```
from CaChannel import ca, CaChannel, CaChannelException
try:
    chan1 = CaChannel('catest')
    chan1.search()
    chan2 = CaChannel('cabo')
    chan2.search()
    chan2.pend_io()
except CaChannelException as e:
    print(e)
```

Write

Multiple channel access write requests.

```
from CaChannel import ca, CaChannel, CaChannelException
try:
    chan1 = CaChannel('catest')
    chan1.search()
    chan2 = CaChannel('cabo')
    chan2.search()
    chan2.pend_io()
    chan1.array_put(1.23)
    chan2.array_put(1)
    chan2.flush_io()
except CaChannelException as e:
    print(e)
```

2.2.3 Asynchronouse Actions

Asynchronous execution does not require that the user wait for completion of an action. Instead, a user specified callback function is executed when the action has completed. Each callback takes two arguments:

- epics_args: arguments returned from epics.
- user_args: arguments specified by the user for use in the callback function.

Since we don't need to wait for actions to complete we use flush_io() instead of pend_io() as in the synchronous examples. Flush_io() starts execution of actions and returns immediately.

Note: The callback function is not executed in the main thread. It runs in an auxiliary thread managed by CA library.

```
import time
from CaChannel import ca, CaChannel, CaChannelException
def connectCB(epics_args, user_args):
    print("connectCB: Python connect callback function")
    print(type(epics_args))
    print(epics_args)
    print(user_args)
    state = epics_args[1]
```

(continues on next page)

(continued from previous page)

```
    if state == ca.CA_OP_CONN_UP:
        print("connectCB: Connection is up")
    elif state == ca.CA_OP_CONN_DOWN:
        print("connectCB: Connection is down")

def putCB(epics_args, user_args):
    print("putCB: Python put callback function")
    print(type(epics_args))
    print(epics_args)
    print(ca.name(epics_args['chid']))
    print(epics_args['type'])
    print(epics_args['count'])
    print(epics_args['status'])
    print(user_args)

chan = CaChannel()
chan.search_and_connect('catest', connectCB)
chan.flush_io()
time.sleep(1)
chan.array_put_callback(3.3, None, None, putCB)
chan.flush_io()
time.sleep(1)
```

2.2.4 Asynchronous Monitoring

Watch for changes in value or alarm state of a process variable. A callback is executed when a change is seen.

```
import sys
import time
from CaChannel import ca, CaChannel, CaChannelException
def eventCB(epics_args, user_args):
    print("eventCb: Python callback function"
    print(type(epics_args))
    print(epics_args)
    print(epics_args['status'])
    print("new value =", epics_args['pv_value'])
    print(epics_args['pv_severity'])
    print(epics_args['pv_status'])

chan = CaChannel()
chan.searchw('catest')
chan.add_masked_array_event(
    ca.DBR_STS_DOUBLE,
    None,
    None,
    eventCB)
chan.flush_io()
time.sleep(5)
```

2.3 Channel Access API

EPICS channel access (CA) is the communication protocol used to transfer information between EPICS servers and clients. Process variables (PV) are accessible though channel access. Interactions with EPICS PV include:

- Connect - create a connection between your application and a PV. This must be done before any other communication with the PV.
- Read - read data (and the meta info, limits, units, precision, statestrings) held in the PV.
- Write - write data to the PV.
- Monitor - notification when a PV's value or alarm state changes.
- Close - close the connection to the PV.

2.3.1 `ca` — Low level interface

This is a module to present the interface of low level channel access C library. It has the same API as module `caffi.ca`.

Data Types

Each PV has a native EPICS type. The native types are then converted to Python types.

This table also lists the EPICS request types. Users can request that the type of the read or write value be changed internally by EPICS. Typically this adds a time penalty and is not recommended.

Native Type	Request Type	C Type	Python Type
`ca.DBF_INT`	`ca.DBR_INT`	16bit short	Integer
`ca.DBF_SHORT`	`ca.DBR_SHORT`	16bit short	Integer
`ca.DBF_LONG`	`ca.DBR_LONG`	32bit int	Integer
`ca.DBF_CHAR`	`ca.DBR_CHAR`	8bit char	Integer
`ca.DBF_STRING`	`ca.DBR_STRING`	array of chars (max 40)	String
`ca.DBF_ENUM`	`ca.DBR_ENUM`	16bit short	Integer
`ca.DBF_FLOAT`	`ca.DBR_FLOAT`	32bit float	Float
`ca.DBF_DOUBLE`	`ca.DBR_DOUBLE`	64bit double	Float

The one area where type conversion is extremely useful is dealing with fields of type *`ca.DBF_ENUM`*. An ENUM value can only be one from a predefined list. A list consists of a set of string values that correspond to the ENUM values (similar to the C enum type). It is easier to remember the list in terms of the strings instead of the numbers corresponding to each string.

Error Code

Error codes defined in header *caerr.h* are supported.

Element Count

Each data field can contain one or more data elements. The number of data elements is referred to as the native element count for a field. The number of data elements written to or read from a data field with multiple elements is user controllable. All or some data elements can be read. When some data elements are accessed the access is always started at the first element. It is not possible to read part of the data and then read the rest of the data.

2.3.2 CaChannel

CaChannel module is a (relatively) high level interface to operate on channel access. It provides almost one to one function map to the channel access C API. So basic knowledge of channel access is assumed.

But it does make it pythonic in other ways, single *CaChannel* object, flexible parameter input and value return.

CaChannel.**USE_NUMPY**
: If numpy support is enabled at compiling time and numpy package is available at runtime, numeric data types can be returned as numpy arrays when *USE_NUMPY=True*. This boosts performance on large size arrays (>1M elements).

Exception CaChannelException

exception CaChannel.**CaChannelException**
: This is the exception type throwed by any channel access operations. Its string representation shows the descriptive message.

Class CaChannel

class CaChannel.**CaChannel**(*pvName=None*)
: CaChannel: A Python class with identical API as of caPython/CaChannel.

  This class implements the methods to operate on channel access so that you can find their C library counterparts , http://www.aps.anl.gov/epics/base/R3-14/12-docs/CAref.html#Function. Therefore an understanding of C API helps much.

  To get started easily, convenient methods are created for often used operations,

Operation	Method
connect	*searchw()*
read	*getw()*
write	*putw()*

  They have shorter names and default arguments. It is recommended to start with these methods. Study the other C alike methods when necessary.

```
>>> chan = CaChannel('catest')
>>> chan.searchw()
>>> chan.putw(12.5)
>>> chan.getw()
```

(continues on next page)

(continued from previous page)

```
12.5
>>> chan.searchw('cabo')
>>> chan.putw('Done')
>>> chan.getw(ca.DBR_STRING)
'Done'
```

Connect

CaChannel.**search**(*pvName=None*)

Attempt to establish a connection to a process variable.

Parameters **pvName** (*bytes, str*) – process variable name

Raises ***CaChannelException*** – if error happens

Note: All remote operation requests such as the above are accumulated (buffered) and not forwarded to the IOC until one of execution methods (*pend_io()*, *poll()*, *pend_event()*, *flush_io()*) is called. This allows several requests to be efficiently sent over the network in one message.

```
>>> chan = CaChannel()
>>> chan.search('catest')
>>> status = chan.pend_io()
>>> chan.state()
<ChannelState.CONN: 2>
```

CaChannel.**search_and_connect**(*pvName*, *callback*, **user_args*)

Attempt to establish a connection to a process variable.

Parameters

- **pvName** (*bytes, str*) – process variable name
- **callback** (*callable*) – function called when connection completes and connection status changes later on.
- **user_args** – user provided arguments that are passed to callback when it is invoked.

Raises ***CaChannelException*** – if error happens

The user arguments are returned to the user in a tuple in the callback function. The order of the arguments is preserved.

Each Python callback function is required to have two arguments. The first argument is a tuple containing the results of the action. The second argument is a tuple containing any user arguments specified by *user_args*. If no arguments were specified then the tuple is empty.

Note: All remote operation requests such as the above are accumulated (buffered) and not forwarded to the IOC until one of execution methods (*pend_io()*, *poll()*, *pend_event()*, *flush_io()*) is called. This allows several requests to be efficiently sent over the network in one message.

```
>>> chan = CaChannel('catest')
>>> def connCB(epicsArgs, _):
...     chid = epicsArgs[0]
...     connection_state = epicsArgs[1]
...     if connection_state == ca.CA_OP_CONN_UP:
...         print(ca.name(chid), "is connected")
>>> chan.search_and_connect(None, connCB, chan)
>>> status = chan.pend_event(3)
catest is connected
>>> chan.search_and_connect('cabo', connCB, chan)
>>> status = chan.pend_event(3)
cabo is connected
>>> chan.clear_channel()
```

CaChannel.**searchw**(*pvName=None*)

Attempt to establish a connection to a process variable.

Parameters **pvName** (*str, None*) – process variable name

Raises ***CaChannelException*** – if timeout or error happens

---

Note: This method waits for connection to be established or fail with exception.

```
>>> chan = CaChannel('non-exist-channel')
>>> chan.searchw()
Traceback (most recent call last):
    ...
CaChannelException: User specified timeout on IO operation expired
```

CaChannel.**clear_channel**()

Close a channel created by one of the search functions.

Clearing a channel does not cause its connection handler to be called. Clearing a channel does remove any monitors registered for that channel. If the channel is currently connected then resources are freed only some time after this request is flushed out to the server.

Note: All remote operation requests such as the above are accumulated (buffered) and not forwarded to the IOC until one of execution methods (*pend_io()*, *poll()*, *pend_event()*, *flush_io()*) is called. This allows several requests to be efficiently sent over the network in one message.

CaChannel.**change_connection_event**(*callback*, **user_args*)

Change the connection callback function

Parameters

- **callback** (*callable*) – function called when connection completes and connection status changes later on. The previous connection callback will be replaced. If an invalid callback is given, no connection callback will be used.
- **user_args** – user provided arguments that are passed to callback when it is invoked.

```
>>> chan = CaChannel('catest')
>>> chan.search() # connect without callback
```

(continues on next page)

(continued from previous page)

```
>>> def connCB(epicsArgs, _):
...     chid = epicsArgs[0]
...     connection_state = epicsArgs[1]
...     if connection_state == ca.CA_OP_CONN_UP:
...         print(ca.name(chid), "is connected")
>>> chan.change_connection_event(connCB) # install connection callback
>>> status = chan.pend_event(3)
catest is connected
>>> chan.change_connection_event(None) # remove connection callback
```

Read

CaChannel.**array_get**(*req_type=None*, *count=None*, ***keywords*)

Read a value or array of values from a channel.

The new value is not available until a subsequent *pend_io()* returns ca.ECA_NORMAL. Then it can be retrieved by a call to *getValue()*.

Parameters

- **req_type** (*int, None*) – database request type (ca.DBR_XXXX). Defaults to be the native data type.
- **count** (*int, None*) – number of data values to read, Defaults to be the native count.
- **keywords** – optional arguments assigned by keywords

key-word	value
use_numpy	True if waveform should be returned as numpy array. Default *CaChannel.USE_NUMPY*.

Raises ***CaChannelException*** – if error happens

Note: All remote operation requests such as the above are accumulated (buffered) and not forwarded to the IOC until one of execution methods (*pend_io()*, *poll()*, *pend_event()*, *flush_io()*) is called. This allows several requests to be efficiently sent over the network in one message.

See also:

getValue()

```
>>> chan = CaChannel('catest')
>>> chan.searchw()
>>> chan.putw(123)
>>> chan.array_get()
>>> chan.pend_io()
>>> chan.getValue()
123.0
```

CaChannel.**getValue**()

Return the value(s) after *array_get()* has completed.

Returns the value returned from the last array_get

Return type int, float, str, list, array, dict

If the *req_type* was not a plain type, the returned value is of dict type. It contains the same keys as in *array_get_callback()*.

See also:

array_get()

```
>>> chan = CaChannel('cabo')
>>> chan.searchw()
>>> chan.putw(1)
>>> chan.array_get(req_type=ca.DBR_CTRL_ENUM)
>>> chan.pend_io()
>>> for k,v in sorted(chan.getValue().items()):
...     print(k, v)
pv_nostrings 2
pv_severity AlarmSeverity.Minor
pv_statestrings ('Done', 'Busy')
pv_status AlarmCondition.State
pv_value 1
```

CaChannel.**array_get_callback**(*req_type*, *count*, *callback*, **user_args*, ***keywords*)

Read a value or array of values from a channel and execute the user supplied callback after the get has completed.

Parameters

- **req_type** (*int, None*) – database request type (ca.DBR_XXXX). Defaults to be the native data type.
- **count** (*int, None*) – number of data values to read, Defaults to be the native count.
- **callback** (*callable*) – function called when the get is completed.
- **user_args** – user provided arguments that are passed to callback when it is invoked.
- **keywords** – optional arguments assigned by keywords

keyword	value
use_numpy	True if waveform should be returned as numpy array. Default *CaChannel.USE_NUMPY*.

Raises ***CaChannelException*** – if error happens

Each Python callback function is required to have two arguments. The first argument is a dictionary containing the results of the action.

field	type	comment	request type				
			DBR_XXXX	DBR_STS_XXXX	DBR_TIME_XXXX	DBR_GR_XXXX	DBR_CTRL_XXXX
chid	int	channels id number	X	X	X	X	X
type	int	database request type (ca.DBR_XXXX)	X	X	X	X	X
count	int	number of values to transfered	X	X	X	X	X
status	int	CA status return code (ca.ECA_XXXX)	X	X	X	X	X
pv_value		PV value	X	X	X	X	X
pv_status	int	PV alarm status		X	X	X	X
pv_severity	int	PV alarm severity		X	X	X	X
pv_seconds	int	seconds part of timestamp			X		
pv_nseconds	int	nanoseconds part of timestamp			X		
pv_nostrings	int	ENUM PV's number of states				X	X
pv_statestrings	string list	ENUM PV's states string				X	X
pv_units	string	units				X	X
pv_precision	int	precision				X	X
pv_updisplim	float	upper display limit				X	X
pv_lodisplim	float	lower display limit				X	X
pv_upalarmlim	float	upper alarm limit				X	X
pv_upwarnlim	float	upper warning limit				X	X
pv_loalarmlim	float	lower alarm limit				X	X
pv_lowarnlim	float	lower warning limit				X	X
pv_upctrllim	float	upper control limit					X
pv_loctrllim	float	lower control limit					X

The second argument is a tuple containing any user arguments specified by *user_args*. If no arguments were specified then the tuple is empty.

Note: All remote operation requests such as the above are accumulated (buffered) and not forwarded to the IOC until one of execution methods (*pend_io()*, *poll()*, *pend_event()*, *flush_io()*) is called. This allows several requests to be efficiently sent over the network in one message.

```
>>> def getCB(epicsArgs, _):
...     for item in sorted(epicsArgs.keys()):
...         if item.startswith('pv_'):
...             print(item,epicsArgs[item])
>>> chan = CaChannel('catest')
>>> chan.searchw()
>>> chan.putw(145)
```

(continues on next page)

(continued from previous page)

```
>>> chan.array_get_callback(ca.DBR_CTRL_DOUBLE, 1, getCB)
>>> status = chan.pend_event(1)
pv_loalarmlim -20.0
pv_loctrllim 0.0
pv_lodislim -20.0
pv_lowarnlim -10.0
pv_precision 4
pv_severity AlarmSeverity.Major
pv_status AlarmCondition.HiHi
pv_units mm
pv_upalarmlim 20.0
pv_upctrllim 0.0
pv_updislim 20.0
pv_upwarnlim 10.0
pv_value 145.0
>>> chan = CaChannel('cabo')
>>> chan.searchw()
>>> chan.putw(0)
>>> chan.array_get_callback(ca.DBR_CTRL_ENUM, 1, getCB)
>>> status = chan.pend_event(1)
pv_nostrings 2
pv_severity AlarmSeverity.No
pv_statestrings ('Done', 'Busy')
pv_status AlarmCondition.No
pv_value 0
```

CaChannel.**getw**(*req_type=None*, *count=None*, ***keywords*)

Read the value from a channel.

If the request type is omitted the data is returned to the user as the Python type corresponding to the native format. Multi-element data has all the elements returned as items in a list and must be accessed using a numerical type. Access using non-numerical types is restricted to the first element in the data field.

`ca.DBF_ENUM` fields can be read using `ca.DBR_ENUM` and `ca.DBR_STRING` types. `ca.DBR_STRING` reads of a field of type `ca.DBF_ENUM` returns the string corresponding to the current enumerated value.

`ca.DBF_CHAR` fields can be read using `ca.DBR_CHAR` and `ca.DBR_STRING` types. `ca.DBR_CHAR` returns a scalar or a sequnece of integers. `ca.DBR_STRING` assumes each integer as a character and assemble a string.

Parameters

- **req_type** (*int, None*) – database request type. Defaults to be the native data type.
- **count** (*int, None*) – number of data values to read, Defaults to be the native count.
- **keywords** – optional arguments assigned by keywords

keyword	value
use_numpy	True if waveform should be returned as numpy array. Default *CaChannel.USE_NUMPY*.

Returns If *req_type* is plain request type, only the value is returned. Otherwise a dict

returns with information depending on the request type, same as the first argument passed to user's callback by *array_get_callback()*.

Raises ***CaChannelException*** – if timeout error happens

```
>>> chan = CaChannel('catest')
>>> chan.searchw()
>>> chan.putw(0)
>>> value = chan.getw(ca.DBR_TIME_DOUBLE)
>>> for k,v in sorted(value.items()):
...     print(k, v)
pv_nseconds ...
pv_seconds ...
pv_severity AlarmSeverity.No
pv_status AlarmCondition.No
pv_value 0.0
```

Changed in version 3.0: If *req_type* is DBR_XXX_STRING for a char type PV, a string will be returned from composing each element as a character.

Write

CaChannel.**array_put**(*value*, *req_type=None*, *count=None*)

Write a value or array of values to a channel

Parameters

- **value** (*int, float, bytes, str, list, tuple, array*) – data to be written. For multiple values use a list or tuple
- **req_type** (*int, None*) – database request type (ca.DBR_XXXX). Defaults to be the native data type.
- **count** (*int, None*) – number of data values to write. Defaults to be the native count.

Note: All remote operation requests such as the above are accumulated (buffered) and not forwarded to the IOC until one of execution methods (*pend_io()*, *poll()*, *pend_event()*, *flush_io()*) is called. This allows several requests to be efficiently sent over the network in one message.

```
>>> chan = CaChannel('catest')
>>> chan.searchw()
>>> chan.array_put(123)
>>> chan.flush_io()
>>> chan.getw()
123.0
>>> chan = CaChannel('cabo')
>>> chan.searchw()
>>> chan.array_put('Busy', ca.DBR_STRING)
>>> chan.flush_io()
>>> chan.getw()
1
>>> chan = CaChannel('cawavec')
>>> chan.searchw()
>>> chan.array_put([1,2,3])
```

(continues on next page)

(continued from previous page)

```
>>> chan.flush_io()
>>> chan.getw()
[1, 2, 3, 0, 0]
>>> chan.getw(count=3, use_numpy=True)
array([1, 2, 3], dtype=uint8)
>>> chan = CaChannel('cawavec')
>>> chan.searchw()
>>> chan.array_put('1234',count=3)
>>> chan.flush_io()
>>> chan.getw(count=4)
[49, 50, 51, 0]
```

CaChannel.**array_put_callback**(*value*, *req_type*, *count*, *callback*, **user_args*)

Write a value or array of values to a channel and execute the user supplied callback after the put has completed.

Parameters

- **value** (*int, float, bytes, str, list, tuple, array*) – data to be written. For multiple values use a list or tuple.
- **req_type** (*int, None*) – database request type (ca.DBR_XXXX). Defaults to be the native data type.
- **count** (*int, None*) – number of data values to write, Defaults to be the native count.
- **callback** (*callable*) – function called when the write is completed.
- **user_args** – user provided arguments that are passed to callback when it is invoked.

Raises *CaChannelException* – if error happens

Each Python callback function is required to have two arguments. The first argument is a dictionary containing the results of the action.

field	type	comment
chid	capsule	channels id structure
type	int	database request type (ca.DBR_XXXX)
count	int	number of values to transfered
status	int	CA status return code (ca.ECA_XXXX)

The second argument is a tuple containing any user arguments specified by *user_args*. If no arguments were specified then the tuple is empty.

Note: All remote operation requests such as the above are accumulated (buffered) and not forwarded to the IOC until one of execution methods (*pend_io()*, *poll()*, *pend_event()*, *flush_io()*) is called. This allows several requests to be efficiently sent over the network in one message.

```
>>> def putCB(epicsArgs, _):
...     print(ca.name(epicsArgs['chid']), 'put completed')
>>> chan = CaChannel('catest')
>>> chan.searchw()
```

(continues on next page)

(continued from previous page)

```
>>> chan.array_put_callback(145, None, None, putCB)
>>> status = chan.pend_event(1)
catest put completed
>>> chan = CaChannel('cabo')
>>> chan.searchw()
>>> chan.array_put_callback('Busy', ca.DBR_STRING, None, putCB)
>>> status = chan.pend_event(1)
cabo put completed
>>> chan = CaChannel('cawave')
>>> chan.searchw()
>>> chan.array_put_callback([1,2,3], None, None, putCB)
>>> status = chan.pend_event(1)
cawave put completed
>>> chan = CaChannel('cawavec')
>>> chan.searchw()
>>> chan.array_put_callback('123', None, None, putCB)
>>> status = chan.pend_event(1)
cawavec put completed
```

CaChannel.**putw**(*value*, *req_type=None*)

Write a value or array of values to a channel.

If the request type is omitted the data is written as the Python type corresponding to the native format. Multi-element data is specified as a tuple or a list. Internally the sequence is converted to a list before inserting the values into a C array. Access using non-numerical types is restricted to the first element in the data field. Mixing character types with numerical types writes bogus results but is not prohibited at this time. `ca.DBF_ENUM` fields can be written using `ca.DBR_ENUM` and `ca.DBR_STRING` types. `ca.DBR_STRING` writes of a field of type `ca.DBF_ENUM` must be accompanied by a valid string out of the possible enumerated values.

Parameters

- **value** (*int, float, bytes, str, tuple, list, array*) – data to be written. For multiple values use a list or tuple
- **req_type** (*int, None*) – database request type (`ca.DBR_XXXX`). Defaults to be the native data type.

Raises ***CaChannelException*** – if timeout or error happens

---

Note: This method does flush the request to the channel access server.

```
>>> chan = CaChannel('catest')
>>> chan.searchw()
>>> chan.putw(145)
>>> chan.getw()
145.0
>>> chan = CaChannel('cabo')
>>> chan.searchw()
>>> chan.putw('Busy', ca.DBR_STRING)
>>> chan.getw()
1
>>> chan.getw(ca.DBR_STRING)
'Busy'
>>> chan = CaChannel('cawave')
>>> chan.searchw()
```

(continues on next page)

(continued from previous page)

```
>>> chan.putw([1,2,3])
>>> chan.getw(req_type=ca.DBR_LONG,count=4)
[1, 2, 3, 0]
>>> chan = CaChannel('cawavec')
>>> chan.searchw()
>>> chan.putw('123')
>>> chan.getw(count=4)
[49, 50, 51, 0]
>>> chan.getw(req_type=ca.DBR_STRING)
'123'
>>> chan = CaChannel('cawaves')
>>> chan.searchw()
>>> chan.putw(['string 1','string 2'])
>>> chan.getw()
['string 1', 'string 2', '']
```

Monitor

CaChannel.**add_masked_array_event**(*req_type*, *count*, *mask*, *callback*, **user_args*, ***keywords*)

Specify a callback function to be executed whenever changes occur to a PV.

Creates a new event id and stores it on self.__evid. Only one event registered per CaChannel object. If an event is already registered the event is cleared before registering a new event.

Parameters

- **req_type** (*int, None*) – database request type (ca.DBR_XXXX). Defaults to be the native data type.
- **count** (*int, None*) – number of data values to read, Defaults to be the native count.
- **mask** (*int, None*) – logical or of ca.DBE_VALUE, ca.DBE_LOG, ca.DBE_ALARM. Defaults to be ca.DBE_VALUE|ca.DBE_ALARM.
- **callback** (*callable*) – function called when the get is completed.
- **user_args** – user provided arguments that are passed to callback when it is invoked.
- **keywords** – optional arguments assigned by keywords

keyword	value
use_numpy	True if waveform should be returned as numpy array. Default *CaChannel.USE_NUMPY*.

Raises ***CaChannelException*** – if error happens

Note: All remote operation requests such as the above are accumulated (buffered) and not forwarded to the IOC until one of execution methods (*pend_io()*, *poll()*, *pend_event()*, *flush_io()*) is called. This allows several requests to be efficiently sent over the network in one message.

```
>>> def eventCB(epicsArgs, _):
...     print('pv_value', epicsArgs['pv_value'])
...     print('pv_status', epicsArgs['pv_status'])
...     print('pv_severity', epicsArgs['pv_severity'])
>>> chan = CaChannel('cabo')
>>> chan.searchw()
>>> chan.putw(1)
>>> chan.add_masked_array_event(ca.DBR_STS_ENUM, None, None, eventCB)
>>> status = chan.pend_event(1)
pv_value 1
pv_status AlarmCondition.State
pv_severity AlarmSeverity.Minor
>>> chan.add_masked_array_event(ca.DBR_STS_STRING, None, None, eventCB)
>>> status = chan.pend_event(1)
pv_value Busy
pv_status AlarmCondition.State
pv_severity AlarmSeverity.Minor
>>> chan.clear_event()
```

CaChannel.**clear_event**()
Remove previously installed callback function.

Note: All remote operation requests such as the above are accumulated (buffered) and not forwarded to the IOC until one of execution methods (*pend_io()*, *poll()*, *pend_event()*, *flush_io()*) is called. This allows several requests to be efficiently sent over the network in one message.

Execute

CaChannel.**pend_io**(*timeout=None*)
Flush the send buffer and wait until outstanding queries (*search()*, *array_get()*) complete or the specified timeout expires.

Parameters **timeout** (*float*) – seconds to wait

Raises ***CaChannelException*** – if timeout or other error happens

CaChannel.**pend_event**(*timeout=None*)
Flush the send buffer and process background activity (connect/get/put/monitor callbacks) for `timeout` seconds.

It will not return before the specified timeout expires and all unfinished channel access labor has been processed.

Parameters **timeout** (*float*) – seconds to wait

Returns `ca.ECA_TIMEOUT`

CaChannel.**poll**()
Flush the send buffer and execute any outstanding background activity.

Returns `ca.ECA_TIMEOUT`

Note: It is an alias to `pend_event(1e-12)`.

CaChannel.**flush_io**()
Flush the send buffer and does not execute outstanding background activity.

Raises *CaChannelException* – if error happens

Information

CaChannel.**field_type**()
Native type of the PV in the server, ca.DBF_XXXX.

```
>>> chan = CaChannel('catest')
>>> chan.searchw()
>>> ftype = chan.field_type()
>>> ftype
<DBF.DOUBLE: 6>
>>> ca.dbf_text(ftype)
'DBF_DOUBLE'
>>> ca.DBF_DOUBLE == ftype
True
```

CaChannel.**element_count**()
Maximum array element count of the PV in the server.

```
>>> chan = CaChannel('catest')
>>> chan.searchw()
>>> chan.element_count()
1
```

CaChannel.**name**()
Channel name specified when the channel was created.

```
>>> chan = CaChannel('catest')
>>> chan.searchw()
>>> chan.name()
'catest'
```

CaChannel.**state**()
Current state of the CA connection.

States	Value	Meaning
ca.cs_never_conn	0	PV not found
ca.cs_prev_conn	1	PV was found but unavailable
ca.cs_conn	2	PV was found and available
ca.cs_closed	3	PV not closed
ca.cs_never_search	4	PV not searched yet

```
>>> chan = CaChannel('catest')
>>> chan.state()
<ChannelState.NEVER_SEARCH: 4>
>>> chan.searchw()
>>> chan.state()
<ChannelState.CONN: 2>
```

CaChannel.**host_name**()
Host name that hosts the process variable.

```
>>> chan = CaChannel('catest')
>>> chan.searchw()
>>> host_name = chan.host_name()
```

CaChannel.**read_access**()
Access right to read the channel.

Returns True if the channel can be read, False otherwise.

```
>>> chan = CaChannel('catest')
>>> chan.searchw()
>>> chan.read_access()
True
```

CaChannel.**write_access**()
Access right to write the channel.

Returns True if the channel can be written, False otherwise.

```
>>> chan = CaChannel('catest')
>>> chan.searchw()
>>> chan.write_access()
True
```

Misc

CaChannel.**setTimeout**(*timeout*)
Set the timeout for this channel object. It overrides the class timeout.

Parameters **timeout** (*float*) – timeout in seconds

```
>>> chan = CaChannel()
>>> chan.setTimeout(10.)
>>> chan.getTimeout()
10.0
```

CaChannel.**getTimeout**()
Retrieve the timeout set for this channel object.

Returns timeout in seconds for this channel instance

Return type float

```
>>> chan = CaChannel()
>>> chan.getTimeout() == CaChannel.ca_timeout
True
```

CaChannel.**replace_access_rights_event**(*callback=None*, *user_args=None*)
Install or replace the access rights state change callback handler for the specified channel.

The callback handler is called in the following situations.

- whenever CA connects the channel immediately before the channel's connection handler is called
- whenever CA disconnects the channel immediately after the channel's disconnect callback is called
- once immediately after installation if the channel is connected

- whenever the access rights state of a connected channel changes

When a channel is created no access rights handler is installed.

Parameters

- **callback** (*callable*) – function called when access rights change. If None is given, remove the access rights event callback.
- **user_args** – user provided arguments that are passed to callback when it is invoked.

```
>>> chan = CaChannel('catest')
>>> chan.searchw()
>>> def accessCB(epicsArgs, _):
...     print('read:', epicsArgs['read_access'], 'write:', epicsArgs[
↪'write_access'])
>>> chan.replace_access_rights_event(accessCB)
read: True write: True
>>> chan.replace_access_rights_event() # clear the callback
```

New in version 3.0.

classmethod CaChannel.**add_exception_event**(*callback=None*, *user_args=None*)

Install or replace the currently installed CA context global exception handler callback.

When an error occurs in the server asynchronous to the clients thread then information about this type of error is passed from the server to the client in an exception message. When the client receives this exception message an exception handler callback is called. The default exception handler prints a diagnostic message on the client's standard out and terminates execution if the error condition is severe.

Note that certain fields returned in the callback args are not applicable in the context of some error messages. For instance, a failed get will supply the address in the client task where the returned value was requested to be written. For other failed operations the value of the addr field should not be used.

Parameters

- **callback** (*callable*) – function called.
- **user_args** – user provided arguments that are passed to callback when it is invoked.

The possible fields available are as defined in the C "struct exception_handler_args" and are: chid, type, count, state, op, ctx, file, lineNo

```
>>> chan = CaChannel('catest')
>>> chan.searchw()
>>> def exceptionCB(epicsArgs, _):
...     print('op:', epicsArgs['op'], 'file:', epicsArgs['file'], 'line:
↪', epicsArgs['lineNo'])
>>> chan.add_exception_event(exceptionCB) # add callback
>>> chan.add_exception_event() # clear the callback
```

New in version 3.1.

classmethod CaChannel.**replace_printf_handler**(*callback=None*, *user_args=None*)

Install or replace the callback used for formatted CA diagnostic message output. The default is to send to stderr.

Parameters

- **callback** (*callable*) – function called.
- **user_args** – user provided arguments that are passed to callback when it is invoked.

```
>>> chan = CaChannel('catest')
>>> chan.searchw()
>>> def printfCB(message, _):
...     print('CA message:', message)
>>> chan.replace_printf_handler(printfCB)  # add callback
>>> chan.replace_printf_handler() # clear the callback
```

New in version 3.1.

2.3.3 epicsPV

This module defines the epicsPV class, which adds additional features to Geoff Savage's CaChannel class.

Author: Mark Rivers Created: Sept. 16, 2002. Modifications:

- Mar. 25, 2014 Xiaoqiang Wang
  - Fix the call sequence inside getCallback
- Mar. 7, 2017 Xiaoqiang Wang
  - Reformat the docstring and code indent.

Class epicsPV

class epicsPV.**epicsPV**(*pvName=None*, *wait=True*)

This class subclasses CaChannel class to add the following features:

- If a PV name is given then the class constructor will do a *CaChannel.CaChannel.searchw()* by default.
- *setMonitor()* sets a generic callback routine for value change events. Subsequent *getw()*, *getValue()* or *array_get()* calls will return the value from the most recent callback, and hence do not result in any network activity or latency. This can greatly improve performance.
- *checkMonitor()* returns a flag to indicate if a callback has occured since the last call to *checkMonitor()*, *getw()*, *getValue()* or *array_get()*. It can be used to increase efficiency in polling applications.
- *getControl()* reads the "control" and other information from an EPICS PV without having to use callbacks. In addition to the PV value, this will return the graphic, control and alarm limits, etc.
- *putWait()* calls *CaChannel.CaChannel.array_put_callback()* and waits for the callback to occur before it returns. This allows programs to wait for record being processed synchronously and without user-written callbacks.

Constructor

epicsPV.**__init__**(*pvName=None*, *wait=True*)
Create an EPICS channel if pvName is specified, and optionally wait for connection.

Parameters

- **pvName** (*str*) – An optional name of an EPICS Process Variable.
- **wait** (*bool*) – If wait is True and pvName is not None then this constructor will do a *CaChannel.CaChannel.searchw()* on the PV. If wait is False and pvName ist not None then this constructor will do a *CaChannel.CaChannel.search()* on the PV, and the user must subsequently do a *CaChannel.CaChannel.pend_io()* on this or another *epicsPV* or *CaChannel* object.

Read

epicsPV.**array_get**(*req_type=None*, *count=None*, ***keywords*)
If *setMonitor()* has not been called then this function simply calls *CaChannel.CaChannel.array_get()*. If *setMonitor()* has been called then it calls *CaChannel.CaChannel.pend_event()* with a very short timeout, and then returns the PV value from the last callback.

epicsPV.**getValue**()
If *setMonitor()* has not been called then this function simply calls *CaChannel.CaChannel.getValue()*. If setMonitor has been called then it calls *CaChannel.CaChannel.pend_event()* with a very short timeout, and then returns the PV value from the last callback.

epicsPV.**getw**(*req_type=None*, *count=None*, ***keywords*)
If *setMonitor()* has not been called then this function simply calls *CaChannel.CaChannel.getw()*. If *setMonitor()* has been called then it calls *CaChannel.CaChannel.pend_event()* with a very short timeout, and then returns the PV value from the last callback.

epicsPV.**getControl**(*req_type=None*, *count=None*, *wait=1*, *poll=0.01*)
Provides a method to read the "control" and other information from an EPICS PV without having to use callbacks.

It calls `CaChannel.CaChannel.CaChannel.array_get_callback()` with a database request type of ca.dbf_type_to_DBR_CTRL(req_type). In addition to the PV value, this will return the graphic, control and alarm limits, etc.

Parameters

- **req_type** (*int*) – request type. Default to field type.
- **count** (*int*) – number of elements. Default to native element count.
- **wait** (*bool*) – If this keyword is 1 (the default) then this routine waits for the callback before returning. If this keyword is 0 then it is the user's responsibility to wait or check for the callback by calling *checkMonitor()*.
- **poll** (*float*) – The timeout for *CaChannel.CaChannel.pend_event()* calls, waiting for the callback to occur. Shorter times reduce the latency at the price of CPU cycles.

```
>>> pv = epicsPV('13IDC:m1')
>>> pv.getControl()
>>> for field in dir(pv.callBack):
...     print field, ':', getattr(pv.callBack, field)
    chid : _bfffec34_chid_p
    count : 1
    monitorState : 0
    newMonitor : 1
    putComplete : 0
    pv_loalarmlim : 0.0
    pv_loctrllim : -22.0
    pv_lodislim : -22.0
    pv_lowarnlim : 0.0
    pv_precision : 4
    pv_riscpad0 : 256
    pv_severity : 0
    pv_status : 0
    pv_units : mm
    pv_upalarmlim : 0.0
    pv_upctrllim : 28.0
    pv_updislim : 28.0
    pv_upwarnlim : 0.0
    pv_value : -15.0
    status : 1
    type : 34
```

Write

epicsPV.**putWait**(*value*, *req_type=None*, *count=None*, *poll=0.01*)

Calls *CaChannel.CaChannel.array_put_callback()* and waits for the callback to occur before it returns. This allows programs to wait for record being processed without having to handle asynchronous callbacks.

Parameters

- **value** – data to be written. For multiple values use a list or tuple
- **req_type** – database request type (ca.DBR_XXXX). Defaults to be the native data type.
- **count** (*int*) – number of data values to write. Defaults to be the native count.
- **poll** (*float*) – The timeout for *CaChannel.CaChannel.pend_event()* calls, waiting for the callback to occur. Shorter times reduce the latency at the price of CPU cycles.

Monitor

epicsPV.**setMonitor**()
Sets a generic callback routine for value change events.

Subsequent *getw()*, *getValue()* or *array_get()* calls will return the value from the most recent callback, do not result in any network latency. This can greatly improve efficiency.

epicsPV.**clearMonitor**()
Cancels the effect of a previous call to *setMonitor()*.

Subsequent *getw()*, *getValue()* or *array_get()* calls will no longer return the value from the most recent callback, but will actually result in channel access calls.

epicsPV.**checkMonitor**()
Returns 1 to indicate if a value callback has occured since the last call to *checkMonitor()*, *getw()*, *getValue()* or *array_get()*, indicating that a new value is available. Returns 0 if no such callback has occurred. It can be used to increase efficiency in polling applications.

2.3.4 epicsMotor

This module provides support for the EPICS motor record.

Author: Mark Rivers

Created: Sept. 16, 2002

Modifications:

- Mar. 7, 2017 Xiaoqiang Wang
  - Reformat the docstring and code indent.
  - Use class property to expose certain fields.

Exception epicsMotorException

exception epicsMotor.**epicsMotorException**
This exception is raised when *epicsMotor.check_limits()* method detects a soft limit or hard limit violation. The *epicsMotor.move()* and *epicsMotor.wait()* methods call *epicsMotor.check_limits()* before they return, unless they are called with the `ignore_limits=True` keyword set.

Class epicsMotor

class epicsMotor.**epicsMotor**(*name*)
This module provides a class library for the EPICS motor record. It uses the *epicsPV.epicsPV* class, which is in turn a subclass of *CaChannel.CaChannel*

Certain motor record fields are exposed as class properties. They can be both read and written unless otherwise noted.

Property	Description	Field
slew_speed	Slew speed or velocity	.VELO
base_speed	Base or starting speed	.VBAS
acceleration	Acceleration time (sec)	.ACCL
description	Description of motor	.DESC
resolution	Resolution (units/step)	.MRES
high_limit	High soft limit (user)	.HLM
low_limit	Low soft limit (user)	.LLM
dial_high_limit	High soft limit (dial)	.DHLM
dial_low_limit	Low soft limit (dial)	.DLLM
backlash	Backlash distance	.BDST
offset	Offset from dial to user	.OFF
done_moving	1=Done, 0=Moving, read-only	.DMOV

```
>>> m = epicsMotor('13BMD:m38')
>>> m.move(10)               # Move to position 10 in user coordinates
>>> m.wait()                 # Wait for motor to stop moving
>>> m.move(50, dial=True)    # Move to position 50 in dial coordinates
>>> m.wait()                 # Wait for motor to stop moving
>>> m.move(1, step=True, relative=True) # Move 1 step relative to current position
>>> m.wait(start=True, stop=True) # Wait for motor to start, then to stop
>>> m.stop()                 # Stop moving immediately
>>> high = m.high_limit      # Get the high soft limit in user coordinates
>>> m.dial_high_limit = 100 # Set the high limit to 100 in dial coodinates
>>> speed = m.slew_speed     # Get the slew speed
>>> m.acceleration = 0.1     # Set the acceleration to 0.1 seconds
>>> val = m.get_position()       # Get the desired motor position in user␣
↪coordinates
>>> dval = m.get_position(dial=True)# Get the desired motor position in dial␣
↪coordinates
>>> rbv = m.get_position(readback=True) # Get the actual position in user␣
↪coordinates
>>> rrbv = m.get_position(readback=True, step=True) # Get the actual motor␣
↪position in steps
>>> m.set_position(100)    # Set the current position to 100 in user coordinates
>>> m.set_position(10000, step=True) # Set the current position to 10000 steps
```

Constructor

epicsMotor.**__init__**(*name*)

Creates a new epicsMotor instance.

Parameters **name** (*str*) – The name of the EPICS motor record without any trailing period or field name.

```
>>> m = epicsMotor('13BMD:m38')
```

Move

epicsMotor.**move**(*value*, *relative=False*, *dial=False*, *step=False*, *ignore_limits=False*)

Moves a motor to an absolute position or relative to the current position in user, dial or step coordinates.

Parameters

- **value** (*float*) – The absolute position or relative amount of the move
- **relative** (*bool*) – If True, move relative to current position. The default is an absolute move.
- **dial** (*bool*) – If True, _value_ is in dial coordinates. The default is user coordinates.
- **step** (*bool*) – If True, _value_ is in steps. The default is user coordinates.
- **ignore_limits** (*bool*) – If True, suppress raising exceptions if the move results in a soft or hard limit violation.

Raises ***epicsMotorException*** – If software limit or hard limit violation detected, unless ignore_limits=True is set.

Note: The "step" and "dial" keywords are mutually exclusive. The "relative" keyword can be used in user, dial or step coordinates.

```
>>> m=epicsMotor('13BMD:m38')
>>> m.move(10)          # Move to position 10 in user coordinates
>>> m.move(50, dial=True)  # Move to position 50 in dial coordinates
>>> m.move(2, step=True, relative=True) # Move 2 steps
```

epicsMotor.**stop**()

Immediately stops a motor from moving by writing 1 to the .STOP field.

```
>>> m=epicsMotor('13BMD:m38')
>>> m.move(10)          # Move to position 10 in user coordinates
>>> m.stop()            # Stop motor
```

epicsMotor.**wait**(*start=False*, *stop=False*, *poll=0.01*, *ignore_limits=False*)

Waits for the motor to start moving and/or stop moving.

Parameters

- **start** (*bool*) – If True, wait for the motor to start moving.
- **stop** (*bool*) – If True, wait for the motor to stop moving.
- **poll** (*float*) – The time to wait between reading the .DMOV field of the record to see if the motor is moving. The default is 0.01 seconds.
- **ignore_limits** (*bool*) – If True, suppress raising an exception if a soft or hard limit is detected.

Raises ***epicsMotorException*** – If software limit or hard limit violation detected, unless *ignore_limits=True* is set.

Note: If neither the "start" nor "stop" keywords are set then "stop" is set to 1, so the routine waits for the motor to stop moving. If only "start" is set to 1 then the routine only waits for the motor to

start moving. If both "start" and "stop" are set to 1 then the routine first waits for the motor to start moving, and then to stop moving.

```
>>> m = epicsMotor('13BMD:m38')
>>> m.move(50)                  # Move to position 50
>>> m.wait(start=True, stop=True)    # Wait for the motor to start moving␣
↪and then to stop moving
```

Readback

epicsMotor.**get_position**(*dial=False*, *readback=False*, *step=False*)

Returns the target or readback motor position in user, dial or step coordinates.

Parameters

- **readback** (*bool*) – If True, return the readback position in the desired coordinate system. The default is to return the target position of the motor.
- **dial** (*bool*) – If True, return the position in dial coordinates. The default is user coordinates.
- **step** (*bool*) – If True, return the position in steps. The default is user coordinates.

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

```
>>> m = epicsMotor('13BMD:m38')
>>> m.move(10)                      # Move to position 10 in user␣
↪coordinates
>>> pos_dial = m.get_position(dial=True)    # Read the target position in␣
↪dial coordinates
>>> pos_step = m.get_position(readback=True, step=True) # Read the␣
↪actual position in steps
```

epicsMotor.**check_limits**()

Check wether there is a soft limit, low hard limit or high hard limit violation.

Raises ***epicsMotorException*** – If software limit or hard limit violation detected.

Config

epicsMotor.**set_position**(*position*, *dial=False*, *step=False*)

Sets the motor position in user, dial or step coordinates.

Parameters

- **position** (*float*) – The new motor position
- **dial** (*bool*) – If True, set the position in dial coordinates. The default is user coordinates.
- **step** (*bool*) – If True, set the position in steps. The default is user coordinates.

Note: The "step" and "dial" keywords are mutually exclusive.

```
>>> m = epicsMotor('13BMD:m38')
>>> m.set_position(10, dial=True)    # Set the motor position to 10 in␣
↪dial coordinates
>>> m.set_position(1000, step=True) # Set the motor position to 1000␣
↪steps
```

2.3.5 CaChannel.util

This module provides functions similiar to those command line tools found in EPICS base, e.g. *caget()*, *caput()*, *camonitor()*, *cainfo()*.

In those functions, *CaChannel.CaChannel* objects are created implicitly and cached in _channel_ dictionary.

```
>>> import time
>>> caput('catest', 1.23, wait=True)
>>> caget('catest')
1.23
>>> caput('cabo', 'Busy')
>>> caget('cabo')
1
>>> caget('cabo', as_string=True)
'Busy'
>>> caput('cawavec', 'this can be a long string')
>>> caget('cawavec', as_string=True)
'this '
>>> caput('cawave', range(4))
>>> caget('cawave', count=4)
[0.0, 1.0, 2.0, 3.0]
```

CaChannel.util.**caget**(*name*, *as_string=False*, *count=None*)

Return PV's current value.

For enum or char type PV, the string form is returned if *as_string* is True. If the PV is of multi-element array, *count* can be used to limit the number of elements.

Parameters

- **name** (*str*) – pv name
- **as_string** (*bool*) – retrieve enum and char type as string
- **count** (*int*) – number of element to request

Returns pv value

CaChannel.util.**caput**(*name*, *value*, *wait=False*, *timeout=None*)

Parameters

- **name** (*str*) – pv name
- **value** – value to write
- **wait** (*bool*) – wait for completion
- **timeout** (*float*) – seconds to wait

CaChannel.util.**camonitor**(*name*, *as_string=False*, *count=None*, *callback=None*)
set a *callback* to be invoked when pv value or alarm status change.

Parameters

- **name** (*str*) – pv name
- **as_string** (*bool*) – retrieve enum and char type as string
- **count** (*int*) – number of element to request
- **callback** – callback function. If *None* is specified, the default callback is to print to the console. If *callback* is not a valid *callable*, any previous callback is removed.

```
>>> camonitor('cacalc')
>>> time.sleep(2)
cacalc ...
>>> def monitor_callback(epics_args, _):
...     for k in sorted(epics_args):
...         print(k, epics_args[k])
>>> camonitor('cacalc', callback=monitor_callback)
>>> time.sleep(2)
chid ...
count 1
pv_nseconds ...
pv_seconds ...
pv_severity AlarmSeverity.No
pv_status AlarmCondition.No
pv_value ...
status ECA.NORMAL
type DBR.TIME_DOUBLE
chid ...
>>> camonitor('cacalc', callback=())
>>> time.sleep(2)
```

CaChannel.util.**cainfo**(*name*)
print pv information

Parameters **name** – pv name

```
>>> caput('cabo', 1)
>>> cainfo('cabo')
cabo
    State:          Connected
    Host:           ...
    Data type:      DBF_ENUM
    Element count:  1
    Access:         RW
    Status:         STATE
    Severity:       MINOR
    Enumerates:     ('Done', 'Busy')
```

2.4 Recipes

2.4.1 Connect Multiple Channels And Get Values

If there are multiple channels to connect, using *searchw()* might not be efficient, because it connects each channel sequentially. A better approach is to create the channels and flush the search request at once.

```
from CaChannel import ca, CaChannel
chans = {pvname: CaChannel(pvname) for pvname in ['pv1', 'pv2', 'pv3', ...]}
for chan in chans.values():
    chan.search()
# call pend_io on either of the channels and the it will flush the requests and wait␣
↪for completion
# if connection does not complete in 10 seconds, CaChannelException is raised with␣
↪status ca.ECA_TIMEOUT
chans['pv1'].pend_io(10)
# if the previous pend_io succeed without exception, we can issue the read request
for chan in chans.values():
    chan.array_get()
# again call pend_io to wait the read requests to succeed or timeout
chans['pv1'].pend_io(10)
# if the previous pend_io succeed without exception, the values can be retrieved with␣
↪getValue
for chan in chans.values():
    print(chan.getValue()
```

2.4.2 Connect Multiple Channels And Monitor Changes

Similiar to the above recipe, but instead of reading the values once, here is to monitor the value changes.

```
from CaChannel import ca, CaChannel

# value change callback
def monitor_callback(epics_arg, user_arg):
    chan = user_arg[0]
    value = epics_arg['pv_value']
    print(chan.name(), value)

# in the connection callback we will subscribe for value changes
def connection_callback(epics_arg, user_arg):
    chan = user_arg[0]
    if epics_arg[1] == ca.CA_OP_CONN_UP:
        chan.add_masked_array_event(None, None, None, monitor_callback, chan)
        chan.flush_io()

# create channels and connect asynchronously
chans = {pvname: CaChannel(pvname) for pvname in ['pv1', 'pv2', 'pv3', ...]}
for chan in chans.values():
    chan.search_and_connect(None, connection_callback, chan)
# flush the channel connection requests
chans['pv1'].flush_io()

# because the callbacks happen in auxiliary threads, the main thread
# is free to do other important stuff, like sleep 10 seconds ;)
time.sleep(10)
```

2.4.3 Get String of Enum

```
from CaChannel import ca, CaChannel
chan = CaChannel('myEnumPV')
chan.searhw()
print(chan.getw(ca.DBR_STRING))
```

2.4.4 Get Control Information

```
from CaChannel import ca, CaChannel
chan = CaChannel('myPV')
chan.searhw()
print(chan.getw(ca.dbf_type_to_DBR_CTRL(chan.field_type())))
```

2.4.5 Get Wavefrom as Numpy Array

- At function level

```
from CaChannel import ca, CaChannel
chan = CaChanne('myWaveformPV')
print(chan.getw(use_numpy=True))
```

- At module level

```
import CaChannel
CaChannel.USE_NUMPY = True
chan = CaChannel.CaChanne('myWaveformPV')
print(chan.getw())
```

2.5 ChangeLog

2.5.1 3.1.3 (01-10-2020)

- Fix various places where conversion exceptions are not handled.
- Improve Python 3 compatibility according to PEP 384.

2.5.2 3.1.2 (29-01-2019)

- Fix epicsPV defaults to wait for connection completion.

2.5.3 3.1.1 (07-12-2018)

- Fix compilation error on Python 3.7.
- Fix compilation error on epics base > 3.14.
- Change to use buffer object instead of numpy/c api to create numpy array.

2.5.4 3.1.0 (15-10-2018)

- Added class methods *CaChannel.CaChannel.add_exception_event()* and *CaChannel.CaChannel.replace_printf_handler()*. They are just thin wrapper over the low level functions `ca.add_exception_event()` and `ca.replace_printf_handler()` respectively.

2.5.5 3.0.4 (15-12-2017)

- Change to link EPICS dynamic libraries if environment variable `EPICS_SHARED` is defined.

2.5.6 3.0.3 (08-12-2017)

- Fix `ca.put()` with non-ascii input string.
- Change that it returns a `bytes` object from non-utf8 C string. It fails with an obscure exception message before.
- Change TravisCI to use conda-forge/linux-anvil docker image, but give the defaults channel higher priority.

2.5.7 3.0.2 (23-10-2017)

- Fix conda build on Linux by pinning conda-build to version 2.

2.5.8 3.0.1 (23-10-2017)

- Allow *count=0* in `ca.get()` if callback is provided.
- Dereference user supplied callbacks - get/put callbacks after being called. - event callback in *CaChannel.CaChannel.clear_event()*.

2.5.9 3.0.0 (06-04-2017)

- Rewrite low level *ca* module with the same API as in package caffi.
- Added method *CaChannel.CaChannel.replace_access_rights_event()*
- Added method *CaChannel.CaChannel.change_connection_event()*
- Added `ca.ECA`, `ca.DBF`, `ca.DBR`, `ca.ChannelState` to represent their C macros `ca.ECA_XXX`, `ca.DBF_XXX`, `ca.DBR_XXX`, `ca.cs_xxx`. For Python < 3.4, this requires module enum34.
- Changed method *CaChannel.CaChannel.getw()* to return string if *req_type* is DBR_STRING for a char waveform.
- Configure continous integration/deployment on Travis/AppVeyor.
- Drop Python 2.4 and 2.5 support.

2.5.10 2.4.2

- Fix chid crash on 64bit windows
- Add epics libs for python 3.5 on windows

2.5.11 2.4.1

- All modules are compatible with Python 2.4+ including Python 3.
- conda build recipe bundle caRepeater program in the package

2.5.12 2.4.0

- Add often used 3rd party module, ca_util, epicsPV and epicsMotor
- Add Anaconda build recipe
- Remove dependency of readline from Com library

2.5.13 2.3.0

- Support Python 3

CHAPTER

THREE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

C

E

INDEX

Symbols

A

C

E

F

G

H

M

N

P

R

S

U

W