caffi Documentation

Release 1.0.3

Xiaoqiang Wang

Aug 04, 2023

Contents

Contents:

CHAPTER 1

Channel Access Guidelines

Note: The original text is from Channel Access Reference Manual But C function names are adapted to Python functions.

1.1 Flushing and Blocking

Significant performance gains can be realized when the CA client library doesn't wait for a response to return from the server after each request. All requests which require interaction with a CA server are accumulated (buffered) and not forwarded to the IOC until one of `ca.flush_io()`, `ca.pend_io()`, `ca.pend_event()`, or `ca.sg_block()` are called allowing several operations to be efficiently sent over the network together.

1.2 Status Codes

If successful, the functions return the status code `ca.ECA.NORMAL`. Unsuccessful status codes returned from the client library are listed with each function.

Operations that appear to be valid to the client can still fail in the server. Writing the string *off* to a floating point field is an example of this type of error. If the server for a channel is located in a different address space than the client then the operations that communicate with the server return status indicating the validity of the request and whether it was successfully enqueued to the server, but communication of completion status is deferred until a user callback is called, or lacking that an exception handler is called.

1.3 User Supplied Callback Functions

Certain CA client initiated requests asynchronously execute an application supplied call back in the client process when a response arrives. The functions `ca.put()`, `ca.get()` and `ca.create_subscription()` all request notification of asynchronous completion via this mechanism.

A dict, *epics_arg* is passed to the application supplied callback. In this dict the *value* field, if present, is any data that might be returned. The *status* field will be set to one of the CA error codes `ca.ECA` and will indicate the status of the operation performed in the IOC. If the status field isn't set to `ca.ECA.NORMAL` or data isn't normally returned from the operation (i.e. put call back) then you should expect that the *value* field will be set to None. The fields *chid* and *type* are set to the values specified when the request was made by the application.

1.4 Channel Access Exceptions

When the server detects a failure, and there is no client call back function attached to the request, an exception handler is executed in the client. The default exception handler prints a message on the console and exits if the exception condition is severe. Certain internal exceptions within the CA client library, and failures detected by the SEVCHK macro may also cause the exception handler to be invoked. To modify this behavior see `ca.add_exception_event()`.

1.5 Server and Client Share the Same Address Space on The Same Host

If the Process Variable's server and it's client are collocated within the same memory address space and the same host then the operations bypass the server and directly interact with the server tool component (commonly the IOC's function block database). In this situation the functions frequently return the completion status of the requested operation directly to the caller with no opportunity for asynchronous notification of failure via an exception handler. Likewise, callbacks may be directly invoked by the CA library functions that request them.

1.6 Arrays

For functions that require an argument specifying the number of array elements, no more than the process variable's maximum native element count may be requested. The process variable's maximum native element count is available from `ca.element_count()` when the channel is connected. If fewer elements than the process variable's native element count are requested, the requested values will be fetched beginning at element zero. By default CA limits the number of elements in an array to be no more than approximately 16k divided by the size of one element in the array. The maximum array size may be configured in the client and in the server, by setting *EPICS_CA_MAX_ARRAY_BYTES*

1.7 Connection Management

Application programs should assume that CA servers may be restarted, and that network connectivity is transient. When you create a CA channel its initial connection state will most commonly be disconnected. If the Process Variable's server is available the library will immediately initiate the necessary actions to make a connection with it. Otherwise, the client library will monitor the state of servers on the network and connect or reconnect with the process variable's server as it becomes available. After the channel connects the application program can freely perform IO operations through the channel, but should expect that the channel might disconnect at any time due to network connectivity disruptions or server restarts.

Three methods can be used to determine if a channel is connected: the application program might call `ca.state()` to obtain the current connection state, block in `ca.pend_io()` until the channel connects, or install a connection callback handler when it calls `ca.create_channel()`. The `ca.pend_io()` approach is best suited to simple command line programs with short runtime duration, and the connection callback method is best suited to toolkit components with long runtime duration. Use of `ca.state()` is appropriate only in programs that prefer to poll

for connection state changes instead of opting for asynchronous notification. The *ca.pend_io()* function blocks only for channels created specifying no callback function. The user's connection state change function will be run immediately from within *ca.create_channel()* if the CA client and CA server are both hosted within the same address space (within the same process).

1.8 Thread Safety and Preemptive Callback to User Code

When the client library is initialized the programmer may specify if preemptive callback is to be enabled. Preemptive callback is disabled by default. If preemptive callback is enabled, then the user's callback functions might be called by CA's auxiliary threads when the main initiating channel access thread is not inside of a function in the channel access client library. Otherwise, the user's callback functions will be called only when the main initiating channel access thread is executing inside of the CA client library. When the CA client library invokes a user's callback function, it will always wait for the current callback to complete prior to executing another callback function. Programmers enabling preemptive callback should be familiar with using mutex locks to create a reliable multi-threaded program. If a GUI toolkit is involved, this means the callback is inside a non GUI thread. Please refer to your GUI toolkits' document, if you want to update GUI inside the callback.

1.9 CA Client Contexts and Application Specific Auxiliary Threads

It is often necessary for several CA client side tools running in the same address space (process) to be independent of each other. For example, the database CA links and the sequencer are designed to not use the same CA client library threads, network circuits, and data structures. Each thread that calls *ca.create_context()* for the first time either directly or implicitly when calling any CA library function for the first time, creates a CA client library context.

A CA client library context contains all of the threads, network circuits, and data structures required to connect and communicate with the channels that a CA client application has created. The priority of auxiliary threads spawned by the CA client library are at fixed offsets from the priority of the thread that called *ca.create_context()*. An application specific auxiliary thread can join a CA context by calling *ca.attach_context()* using the CA context identifier that was returned from *ca.current_context()* when it is called by the thread that created the context which needs to be joined. A context which is to be joined must be preemptive - it must be created using *create_context(True)*. It is not possible to attach a thread to a non-preemptive CA context created implicitly or explicitly with *create_context(False)*. Once a thread has joined with a CA context it need only make ordinary function calls to use the context.

A CA client library context can be shut down and cleaned up, after destroying any channels or application specific threads that are attached to it, by calling *ca.destroy_context()*. The context may be created and destroyed by different threads as long as they are both part of the same context.

1.10 Polling the CA Client Library From Single Threaded Applications

If preemptive call back is not enabled, then for proper operation CA must periodically be polled to take care of background activity. This requires that your application must either wait in one of *ca.pend_event()*, *ca.pend_io()*, or *ca.sg_block()* or alternatively it must call *ca.poll()* at least every 100 milli-seconds.

CHAPTER 2

Module `caffi.ca`

The is the low level access to EPICS channel access library. It maps the corresponding C API to Python functions. Even though as same as possible, there are subtle differences:

- The *ca_* prefix of the C function name has been removed, e.g. C function *ca_create_channel* is now Python function *create_channel()*.
- The C function *ca_context_destroy* has been renamed to *destroy_context()* to have the same symmetry as *XXX*_context.
- The two separate C functions *ca_client_status* and *ca_context_status* have been merged into *show_context()*. Internally it calls the appropriate C function depending on whether *context* is given.
- The order of the argument might have been altered to allow default arguments, thus more pythonic. Take C function *array_put_callback* for example,

```
int ca_array_put_callback
(
    chtype                  type,
    unsigned long           count,
    chid                    chanId,
    const void *            pValue,
    caEventCallBackFunc *   pFunc,
    void *                  pArg
);
```

  Its Python counterpart,

```
put(chid, value, chtype=None, count=None, callback=None)
```

  Only two arguments are mandatory, which are the channel identifier and the value to write. The others are made optional and have reasonable defaults.

- In C API the callback function handler has the following signature,

```
void (*)(struct xxx_handler_args)
```

In Python counterpart, the callback signature is

```
def callback(epicsArgs):
```

epicsArgs is a dict converted from *xxx_handler_args*.

- C functions normally return status code to indicate success or failure. The Python counterparts follow this but have exceptions:
  - The get functions in C require a user supplied memory pointer passed as argument. In Python the function *get()* and *sg_get()* returns a tuple of form (*ECA*, *DBRValue*)

    The first item is the status code, which must be *ECA.NORMAL*, before using the second item.

    The second item holds the reference to the allocated memory. And the memory is not stable until a subsequent call of *pend_io()* returns *ECA.NORMAL*. Then the value can be retrieved by calling *DBRValue.get()*.
  - All the creation functions, *create_channel()*, *create_subscription()* and *sg_create()* return a tuple of the form (*ECA*, *object identifier*). The object identifier can only be used if the first item is *ECA.NORMAL*.
- In C the following macros definition are also accessible as `enum.IntEnum` type:

C Macros	Python Enum
ECA_XXX	*ECA*
DBE_XXX	*DBE*
DBF_XXX	*DBF*
DBR_XXX	*DBR*
CA_OP_XXX	*CA_OP*
cs_xxx	*ChannelState*
XXX_ALARM (severity)	*AlarmSeverity*
XXX_ALARM (status)	*AlarmCondition*

This makes it convenient when interactively examine the code value. e.g.

```
>>> ca.create_context(True)
<ECA.NORMAL: 1>
>>> status, chid = ca.create_channel('catest')
>>> ca.pend_io(2)
<ECA.NORMAL: 1>
>>> ca.name(chid)
'catest'
>>> ca.host_name(chid)
'localhost:5064'
>>> ca.field_type(chid)
<DBF.DOUBLE: 6>
>>> ca.element_count(chid)
1
>>> ca.read_access(chid)
True
>>> ca.write_access(chid)
True
>>> ca.put(chid, 10)
<ECA.NORMAL: 1>
```

(continues on next page)

(continued from previous page)

```
>>> ca.flush_io()
<ECA.NORMAL: 1>
>>> status, dbrvalue = ca.get(chid)
>>> ca.pend_io(2)
<ECA.NORMAL: 1>
>>> dbrvalue.get()
10.0
```

2.1 Context

caffi.ca.**create_context**(*preemptive_callback=True*)

Parameters **preemptive_callback** (*bool*) – enable preemptive callback

Returns

- *ECA.NORMAL* - Normal successful completion
- *ECA.ALLOCMEM* - Failed, unable to allocate space in pool
- ***ECA.NOTTHREADED*** **- Current thread is already a member of a non-preemptive callback CA context** (possibly created implicitly)

This function, or *attach_context()*, should be called once from each thread prior to making any of the other Channel Access calls. If one of the above is not called before making other CA calls then a non-preemptive context is created by default, and future attempts to create a preemptive context for the current threads will fail.

If preemptive callback is disabled then additional threads are not allowed to join the CA context using *attach_context()* because allowing other threads to join implies that CA callbacks will be called preemptively from more than one thread.

caffi.ca.**destroy_context**()

Shut down the calling thread's channel access client context and free any resources allocated. Detach the calling thread from any CA client context.

Any user-created threads that have attached themselves to the CA context must stop using it prior to its being destroyed.

On many OS that execute programs in a process based environment the resources used by the client library such as sockets and allocated memory are automatically released by the system when the process exits and *destroy_context()* hasn't been called, but on light weight systems such as vxWorks or RTEMS no cleanup occurs unless the application calls *destroy_context()*.

caffi.ca.**attach_context**(*context*)

Parameters **context** (*cdata*) – The CA context to join with.

Returns

- *ECA.NORMAL* - Normal successful completion
- *ECA.NOTTHREADED* - Context is not preemptive so cannot be joined
- *ECA.ISATTACHED* - Thread already attached to a CA context

The calling thread becomes a member of the specified CA context. If context is non preemptive, then additional threads are not allowed to join the CA context because allowing other threads to join implies that CA callbacks will be called preemptively from more than one thread.

caffi.ca.**detach_context**()
Detach from any CA context currently attached to the calling thread.

This does not cleanup or shutdown any currently attached CA context.

caffi.ca.**current_context**()

Returns The current thread's CA context. If none then None is returned.

caffi.ca.**show_context**(*context=None*, *level=0*)
Prints information about the client context including, at higher interest levels, status for each channel.

Parameters

- **context** (*cdata, None*) – The CA context to examine. Default is the calling threads CA context.
- **level** (*int*) – The interest level. Increasing level produces increasing detail.

2.2 Channel

caffi.ca.**create_channel**(*name*, *callback=None*, *priority=<CA_PRIORITY.MIN: 0>*)
This function creates a CA channel.

Parameters

- **name** (*str*) – Process variable name string.
- **callback** (*callable, None*) – Optional user's call back function to be run when the connection state changes. Casual users of channel access may decide to leave it None if they do not need to have a call back function run in response to each connection state change event. The callback receives one *dict* argument including the following fields:

field	value
chid	channel identifier
op	– CA_OP.CONN_UP - connected – CA_OP.CONN_DOWN - disconnected

- **priority** (int, *CA_PRIORITY*) – The priority level for dispatch within the server or network, with 0 specifying the lowest dispatch priority and 99 the highest. This parameter currently does not impact dispatch priorities within the client, but this might change in the future. The abstract priority range specified is mapped into an operating system specific range of priorities within the server. This parameter is ignored if the server is running on a network or operating system that does not have native support for prioritized delivery or execution respectively. Specifying many different priorities within the same program can increase resource consumption in the client and the server because an independent virtual circuit, and associated data structures, is created for each priority that is used on a particular server.

Returns

(*ECA*, channel identifier or None)

- *ECA.NORMAL* - Normal successful completion
- *ECA.BADSTR* - Invalid string

- *ECA.BADPRIORITY* - Invalid priority
- *ECA.UNAVAILINSERV* - Not supported by attached service
- *ECA.ALLOCMEM* - Unable to allocate memory

The CA client library will attempt to establish and maintain a virtual circuit between the caller's application and a named process variable in a CA server. Each call to ca_create_channel allocates resources in the CA client library and potentially also a CA server. The function *clear_channel()* is used to release these resources.

If successful, the routine returns a channel identifier. This identifier can be used with any channel access call that operates on a channel.

The circuit may be initially connected or disconnected depending on the state of the network and the location of the channel. A channel will only enter a connected state after the server's address is determined, and only if channel access successfully establishes a virtual circuit through the network to the server. Channel access routines that send a request to a server will return *ECA.DISCONNCHID* if the channel is currently disconnected.

There are two ways to obtain asynchronous notification when a channel enters a connected state.

- The first and simplest method requires that you call ca_pend_io(), and wait for successful completion, prior to using a channel that was created without specifying call back function.
- The second method requires that you register a connection handler by supplying a valid connection call-back function pointer. This connection handler is called whenever the connection state of the channel changes. If you have installed a connection handler then *pend_io()* will not block waiting for the channel to enter a connected state.

The function *state()* can be used to test the connection state of a channel. Valid connections may be isolated from invalid ones with this function *pend_io()* times out.

Due to the inherently transient nature of network connections the order of connection call backs relative to the order that *create_channel()* calls are made by the application can't be guaranteed, and application programs may need to be prepared for a connected channel to enter a disconnected state at any time.

caffi.ca.**clear_channel**(*chid*)

Shutdown and reclaim resources associated with a channel created by ca_create_channel().

Parameters **chid** (*cdata*) – Channel identifier

Returns

- *ECA.NORMAL* - Normal successful completion
- *ECA.BADCHID* - Corrupted CHID

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

Clearing a channel does not cause its disconnect handler to be called, but clearing a channel does shutdown and reclaim any channel state change event subscriptions (monitors) registered with the channel.

caffi.ca.**change_connection_event**(*chid*, *callback=None*)

Change the connection event callback function.

Parameters

- **chid** – Channel identifier
- **callback** – User's call back function to be run when the connection state changes. The callback receives the same argument as *create_channel()*. This will replace the previous connection callback function. If an invalid *callback* is given, no connection callback is used.

Returns

- *ECA.NORMAL* - Normal successful completion
- *ECA.BADCHID* - Corrupted CHID

2.3 Operation

caffi.ca.**create_subscription**(*chid*, *callback*, *chtype=None*, *count=None*, *mask=None*, *use_numpy=False*)

Register a state change subscription and specify a call back function to be invoked whenever the process variable undergoes significant state changes.

Parameters

- **chid** (*cdata*) – Channel identifier
- **callback** (*callable*) – User supplied callback function to be run when requested operation completes. The callback receives one *dict* argument including the following fields:

field	value
chid	channel identifier
type	the type of the item returned, *DBR*
count	the element count of the item returned
status	status code of the request from the server, *ECA*
value	If *type* is a plain type, this is the PV's value. Otherwise it is a dict containing the meta information associated with this *type*.

- **chtype** (*DBR*, None) – The external type of the supplied value to be written. Conversion will occur if this does not match the native type. Default is the native type.
- **count** (*int, None*) – Element count to be written to the channel. Default is native element count.
- **mask** (*DBE*, None) – A mask with bits set for each of the event trigger types requested. The event trigger mask must be a bitwise or of one or more of *DBE*.
- **use_numpy** (*bool*) – whether to format numeric waveform as numpy array

Returns

(*ECA*, event identifier or None)

- *ECA.NORMAL* - Normal successful completion
- *ECA.BADCHID* - Corrupted CHID
- *ECA.BADTYPE* - Invalid DBR_XXXX type
- *ECA.ALLOCMEM* - Unable to allocate memory
- *ECA.ADDFAIL* - A local database event add failed

A significant change can be a change in the process variable's value, alarm status, or alarm severity. In the process control function block database the deadband field determines the magnitude of a significant change for the process variable's value. Each call to this function consumes resources in the client library and potentially a CA server until one of *clear_channel()* or *clear_subscription()* is called.

Subscriptions may be installed or canceled against both connected and disconnected channels. The specified *callback* is called once immediately after the subscription is installed with the process variable's current state if the process variable is connected. Otherwise, the specified *callback* is called immediately after establishing a connection (or reconnection) with the process variable. The specified *callback* is called immediately with the process variable's current state from within *create_subscription()* if the client and the process variable share the same address space.

If a subscription is installed on a channel in a disconnected state then the requested count will be set to the native maximum element count of the channel if the requested count is larger.

All subscription requests such as the above are accumulated (buffered) and not forwarded to the IOC until one of *flush_io()*, *pend_io()*, or *pend_event()* are called. This allows several requests to be efficiently sent over the network in one message.

If at any time after subscribing, read access to the specified process variable is lost, then the call back will be invoked immediately indicating that read access was lost via the status argument. When read access is restored normal event processing will resume starting always with at least one update indicating the current state of the channel.

caffi.ca.**clear_subscription**(*evid*)

Cancel a subscription.

Parameters **evid** (*cdata*) – event id returned by *create_subscription()*

Returns

- ECA.NORMAL - Normal successful completion
- ECA.BADCHID - Corrupted CHID

All cancel-subscription requests such as the above are accumulated (buffered) and not forwarded to the server until one of *flush_io()*, *pend_io()*, or *pend_event()* are called. This allows several requests to be efficiently sent together in one message.

caffi.ca.**get**(*chid*, *chtype=None*, *count=None*, *callback=None*, *use_numpy=False*)

Read a scalar or array value from a process variable.

Parameters

- **chid** (*cdata*) – Channel identifier
- **chtype** (int, *DBR*, None) – The external type of the supplied value to be written. Conversion on the server will occur if this does not match the native type. Default is the native type.
- **count** (*int, None*) – Element count to be read from the specified channel. If *callback* is specified, a count of zero means use the current element count from the server.
- **callback** (*callable, None*) – User supplied callback function to be run when requested operation completes. The callback receives one *dict* argument including the following fields:

field	value
chid	channel identifier
type	the type of the item returned, *DBR*
count	the element count of the item returned
status	status code of the request from the server, *ECA*
value	If *type* is a plain type, this is the PV's value. Otherwise it is a dict containing the meta information associated with this *type*.

- **use_numpy** (*bool*) – whether to format numeric waveform as numpy array

Returns

(*ECA*, *DBRValue* or None)

- *ECA.NORMAL* - Normal successful completion
- *ECA.BADTYPE* - Invalid DBR_XXXX type
- *ECA.BADCHID* - Corrupted CHID
- *ECA.BADCOUNT* - Requested count larger than native element count
- *ECA.GETFAIL* - A local database get failed
- ECA.NORDACCESS - Read access denied
- *ECA.ALLOCMEM* - Unable to allocate memory
- *ECA.DISCONN* - Channel is disconnected

When no *callback* is specified, call *DBRValue.get()* to retrieve the value only if *ECA.NORMAL* is returned from a subsequent *pend_io()*. If a connection is lost outstanding ca get requests are not automatically reissued following reconnect.

When *callback* is specified a value is read from the channel and then the user's callback is invoked with a dict containing the value. Note that ca_pend_io will not block for the delivery of values. If the channel disconnects before a ca get callback request can be completed, then the clients call back function is called with failure status.

All of these functions return *ECA.DISCONN* if the channel is currently disconnected.

All get requests are accumulated (buffered) and not forwarded to the IOC until one of *flush_io()*, *pend_io()*, or *pend_event()* are called. This allows several requests to be efficiently sent over the network in one message.

caffi.ca.**put**(*chid*, *value*, *chtype=None*, *count=None*, *callback=None*)

Write a scalar or array value to a process variable.

Parameters

- **chid** (*cdata*) – Channel identifier
- **value** (*int, float, bytes, str, tuple, list, array*) – A scalar or array value to be written to the channel. If *value* is of string type, it will first be convert to bytes using UTF8 codec. And the following conversion may be involved:

request type	conversion
DBR.STRING	nothing
DBR.ENUM	request type is changed to DBR.STRING
DBR.CHAR	a list of byte integers
Other types	a float number

- **chtype** (int, *DBR*, None) – The external type of the supplied value to be written. Conversion on the server will occur if this does not match the native type. Default is the native type.
- **count** (*int, None*) – Element count to be written to the channel. Default is native element count. But it can be reduced to match the length of user supplied value.
- **callback** (*callable, None*) – User supplied callback function to be run when requested operation completes. The callback receives one *dict* argument including the following fields:

field	value
chid	channel identifier
type	DBR.INVALID (-1)
count	0
status	status code of the request from the server, *ECA*

Returns

- *ECA.NORMAL* - Normal successful completion
- *ECA.BADCHID* - Corrupted CHID
- *ECA.BADTYPE* - Invalid DBR_XXXX type
- *ECA.BADCOUNT* - Requested count larger than native element count
- *ECA.STRTOBIG* - Unusually large string supplied
- *ECA.NOWTACCESS* - Write access denied
- *ECA.ALLOCMEM* - Unable to allocate memory
- *ECA.DISCONN* - Channel is disconnected

When invoked without callback the client will receive no response unless the request can not be fulfilled in the server. If unsuccessful an exception handler is run on the client side.

When invoked with callback the user supplied asynchronous call back is called only after the initiated write operation, and all actions resulting from the initiating write operation, complete.

If unsuccessful the call back function is invoked indicating failure status.

If the channel disconnects before a put callback request can be completed, then the client's call back function is called with failure status, but this does not guarantee that the server did not receive and process the request before the disconnect. If a connection is lost and then resumed outstanding ca put requests are not automatically reissued following reconnect.

All of these functions return *ECA.DISCONN* if the channel is currently disconnected.

All put requests are accumulated (buffered) and not forwarded to the IOC until one of *flush_io()*, *pend_io()*, or *pend_event()* are called. This allows several requests to be efficiently combined into one message.

2.4 Execution

caffi.ca.**pend**(*timeout*, *early*)

Parameters

- **timeout** (*float*) – Specifies the time out interval. A timeout interval of zero specifies forever.
- **early** (*bool*) – Call *pend_io()* if *early* is True otherwise *pend_event()* is called

Returns

- *ECA.NORMAL* - Normal successful completion
- *ECA.TIMEOUT* - Selected IO requests didn't complete before specified timeout
- *ECA.EVDISALLOW* - Function inappropriate for use within an event handler

caffi.ca.**pend_io**(*timeout*)

This function flushes the send buffer and then blocks until outstanding get requests *without* callback complete, and until channels created *without* callback connect for the first time.

Parameters **timeout** (*float*) – Specifies the time out interval. A timeout interval of zero specifies forever.

Returns

- *ECA.NORMAL* - Normal successful completion
- *ECA.TIMEOUT* - Selected IO requests didn't complete before specified timeout
- *ECA.EVDISALLOW* - Function inappropriate for use within an event handler

If *ECA.NORMAL* is returned then it can be safely assumed that all outstanding get requests *without* callback have completed successfully and channels created *without* callback have connected for the first time.

If *ECA.TIMEOUT* is returned then it must be assumed for all previous get requests and properly qualified first time channel connects have failed.

If *ECA.TIMEOUT* is returned then get requests may be reissued followed by a subsequent call to *pend_io()*. Specifically, the function will block only for outstanding get requests issued, and also any channels created *without* callback, after the last call to ca_pend_io() or ca client context creation whichever is later. Note that *create_channel()* requests generally should not be reissued for the same process variable unless *clear_channel()* is called first.

If no get or connection state change events are outstanding then *pend_io()* will flush the send buffer and return immediately without processing any outstanding channel access background activities.

The delay specified should take into account worst case network delays such as Ethernet collision exponential back off until retransmission delays which can be quite long on overloaded networks.

Unlike *pend_event()*, this routine will not process CA's background activities if none of the selected IO requests are pending.

caffi.ca.**pend_event**(*timeout*)

The send buffer is flushed and CA background activity is processed for *timeout* seconds.

Parameters **timeout** (*float*) – The duration to block in this routine in seconds. A timeout of zero seconds blocks forever.

Returns

- *ECA.TIMEOUT* - The operation timed out
- *ECA.EVDISALLOW* - Function inappropriate for use within a call back handler

The *pend_event()* function will not return before the specified timeout expires and all unfinished channel access labor has been processed, and unlike *pend_io()* returning from the function does not indicate anything about the status of pending IO requests.

It return *ECA.TIMEOUT* when successful. This behavior probably isn't intuitive, but it is preserved to insure backwards compatibility.

See also Thread Safety and Preemptive Callback to User Code.

caffi.ca.**poll**()

The send buffer is flushed and any outstanding CA background activity is processed.

Note: same as pend_event(1e-12)

---

caffi.ca.**flush_io**()
Flush outstanding IO requests to the server.

Returns

- *ECA.NORMAL* - Normal successful completion

This routine might be useful to users who need to flush requests prior to performing client side labor in parallel with labor performed in the server. Outstanding requests are also sent whenever the buffer which holds them becomes full.

caffi.ca.**test_io**()
This function tests to see if all get requests are complete and channels created without a connection callback function are connected. It will report the status of outstanding get requests issued, and channels created without connection callback function, after the last call to ca_pend_io() or CA context initialization whichever is later.

Returns

- *ECA.IODONE* - All IO operations completed
- *ECA.IOINPROGRESS* - IO operations still in progress

2.5 Information

caffi.ca.**field_type**(*chid*)

Parameters **chid** (*cdata*) – channel identifier

Returns the native type in the server of the process variable.

Return type *DBF*

caffi.ca.**element_count**(*chid*)

Parameters **chid** (*cdata*) – channel identifier

Returns the maximum array element count in the server for the specified IO channel.

caffi.ca.**name**(*chid*)

Parameters **chid** (*cdata*) – channel identifier

Returns the name provided when the supplied channel id was created.

caffi.ca.**state**(*chid*)

Parameters **chid** (*cdata*) – channel identifier

Returns the connection state

Return type *ChannelState*

caffi.ca.**message**(*status*)

Parameters **status** (*int*) – CA status code

Returns a message corresponding to a user specified CA status code.

caffi.ca.**host_name**(*chid*)

Parameters **chid** – channel identifier

Returns the name of the host to which a channel is currently connected.

caffi.ca.**read_access**(*chid*)

Parameters **chid** – channel identifier

Returns True if the client currently has read access to the specified channel and False otherwise.

caffi.ca.**write_access**(*chid*)

Parameters **chid** – channel identifier

Returns True if the client currently has write access to the specified channel and False otherwise.

2.6 Synchronous

caffi.ca.**sg_create**()

Create a synchronous group and return an identifier for it.

Returns (*ECA*, int or None)

A synchronous group can be used to guarantee that a set of channel access requests have completed. Once a synchronous group has been created then channel access get and put requests may be issued within it using *sg_get()* and *sg_put()* respectively. The routines *sg_block()* and *sg_test()* can be used to block for and test for completion respectively. The routine *sg_reset()* is used to discard knowledge of old requests which have timed out and in all likelihood will never be satisfied.

Any number of asynchronous groups can have application requested operations outstanding within them at any given time.

caffi.ca.**sg_delete**(*gid*)

Deletes a synchronous group.

Parameters **gid** (*int*) – Identifier of the synchronous group to be deleted.

Returns

- *ECA.NORMAL* - Normal successful completion
- *ECA.BADSYNCGRP* - Invalid synchronous group

caffi.ca.**sg_get**(*gid*, *chid*, *chtype=None*, *count=None*, *use_numpy=False*)

Read a value from a channel and increment the outstanding request count of a synchronous group.

Parameters

- **gid** (*int*) – Identifier of the synchronous group.
- **chid** (*cdata*) – Channel identifier
- **chtype** (int, *DBR*, None) – External type of returned value. Conversion on the server will occur if this does not match native type.
- **count** (*int, None*) – Element count to be read from the specified channel.
- **use_numpy** – whether to format numeric waveform as numpy array

Returns

(*ECA*, *DBRValue* or None)

- *ECA.NORMAL* - Normal successful completion
- *ECA.BADSYNCGRP* - Invalid synchronous group
- *ECA.BADCHID* - Corrupted CHID
- *ECA.BADCOUNT* - Requested count larger than native element count

- *ECA.BADTYPE* - Invalid DBR_XXXX type
- *ECA.GETFAIL* - A local database get failed

Call *DBRValue.get()* to retrieve the value only if *ECA.NORMAL* has been received from ca_sg_block , or until *sg_test()* returns True.

All remote operation requests such as the above are accumulated (buffered) and not forwarded to the server until one of *flush_io()*, *pend_io()*, or *pend_event()* are called. This allows several requests to be efficiently sent in one message.

If a connection is lost and then resumed outstanding gets are not reissued.

caffi.ca.**sg_put**(*gid*, *chid*, *value*, *chtype=None*, *count=None*)

Write a value, or array of values, to a channel and increment the outstanding request count of a synchronous group.

Parameters

- **gid** (*int*) – Synchronous group identifier
- **chid** (*cdata*) – Channel identifier
- **value** (*int, float, bytes, str, tuple, list, array*) – The value or array of values to write. If *value* is of string type, it will first be convert to bytes using UTF8 codec. And the following conversion may be involved:

request type	conversion
DBR.STRING	nothing
DBR.ENUM	request type is changed to DBR.STRING
DBR.CHAR	a list of byte integers
Other types	a float number

- **chtype** (int, *DBR*) – The type of supplied value. Conversion on the server will occur if it does not match the native type.
- **count** – The element count to be written to the specified channel.

Returns

- *ECA.NORMAL* - Normal successful completion
- *ECA.BADSYNCGRP* - Invalid synchronous group
- *ECA.BADCHID* - Corrupted CHID
- *ECA.BADTYPE* - Invalid DBR_XXXX type
- *ECA.BADCOUNT* - Requested count larger than native element count
- *ECA.STRTOBIG* - Unusually large string supplied
- *ECA.PUTFAIL* - A local database put failed

All remote operation requests such as the above are accumulated (buffered) and not forwarded to the server until one of *flush_io()*, *pend_io()* or *pend_event()* are called. This allows several requests to be efficiently sent in one message.

If a connection is lost and then resumed outstanding puts are not reissued.

caffi.ca.**sg_block**(*gid*, *timeout*)

Flushes the send buffer and then waits until outstanding requests complete or the specified time out expires. At this time outstanding requests include calls to *sg_get()* and calls to *sg_put()*. If *ECA.TIMEOUT* is returned then failure must be assumed for all outstanding queries. Operations can be reissued followed

by another *sg_block()*. This routine will only block on outstanding queries issued after the last call to *sg_block()*, *sg_reset()*, or *sg_create()* whichever occurs later in time. If no queries are outstanding then *sg_block()* will return immediately without processing any pending channel access activities.

Values written into your program's variables by a channel access synchronous group request should not be referenced by your program until *ECA.NORMAL* has been received from *sg_block()*. This routine will process pending channel access background activity while it is waiting.

Parameters

- **gid** (*int*) – Identifier of the synchronous group.
- **timeout** (*float*) – Specifies the time out interval. A timeout interval of zero specifies forever.

Returns

- *ECA.NORMAL* - Normal successful completion
- *ECA.TIMEOUT* - The operation timed out
- *ECA.EVDISALLOW* - Function inappropriate for use within an event handler
- *ECA.BADSYNCGRP* - Invalid synchronous group

caffi.ca.**sg_test**(*gid*)

Test to see if all requests made within a synchronous group have completed.

Parameters **gid** (*int*) – Identifier of the synchronous group.

Returns

- *ECA.IODONE* - IO operations completed
- *ECA.IOINPROGRESS* - Some IO operations still in progress

caffi.ca.**sg_reset**(*gid*)

Reset the number of outstanding requests within the specified synchronous group to zero so that *sg_test()* will return True and *sg_block()* will not block unless additional subsequent requests are made.

Parameters **gid** (*int*) – Identifier of the synchronous group.

Returns

- *ECA.NORMAL* - Normal successful completion
- *ECA.BADSYNCGRP* - Invalid synchronous group

2.7 Misc

caffi.ca.**add_exception_event**(*callback=None*)

Replace the currently installed CA context global exception handler call back.

Parameters **callback** (*callable, None*) – User callback function to be executed when an exceptions occur. Passing None causes the default exception handler to be reinstalled. The argument is a dict including the following fields:

field	value
chid	channel identifier (may be NULL)
type	type requested
count	count requested
addr	user's address to write results of CA_OP.GET (may be NULL)
stat	status code, *ECA*
op	operation, *CA_OP*
ctx	a character string containing context info
file	source file name (may be empty)
lineNo	source file line number (may be zero)

Returns

- *ECA.NORMAL* - Normal successful completion

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: Certain fields 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.

---

caffi.ca.**replace_access_rights_event**(*chid*, *callback=None*)

Install or replace the access rights state change callback handler for the specified channel.

Parameters

- **chid** (*cdata*) – The channel identifier
- **callback** (*callable, None*) – User supplied call back function. Passing None uninstalls the current handler. The callback receives one *dict* argument including the following fields:

field	value
chid	channel identifier
read_access	True if with read access rights
write_access	True if with write access rights

Returns

- *ECA.NORMAL* - Normal successful completion
- *ECA.BADCHID* - Corrupted CHID

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 call back 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.

class caffi.ca.**DBRValue**(*dbrtype=<DBR.INVALID: -1>*, *count=0*, *cvalue=<cdata 'void *' NULL>*, *use_numpy=False*)

Parameters

- **dbrtype** – The external type of the supplied *cvalue*
- **count** – Element count of the supplied *cvalue*
- **cvalue** – Pointer to the structure of *dbrtype* with *count* element
- **use_numpy** (*bool*) – whether to format numeric waveform as numpy array

An convenient object to represent the value returned by *caffi.ca.get()* and *caffi.ca.sg_get()*. It holds the reference to the memory allocated by the get functions, in addition the type and element count information.

Once the memory is assured to be stable, normally when the gets function completed with success, call *get()* to get the returned values.

get()

Returns Value for plain DBR_XXXX type or a dict for DBR_STS_XXXX etc.

Note: This method should be called only if the get request has succeeded.

2.8 Constants

class caffi.ca.**ECA**

Enum redefined from ECA_XXX status code

Note: ARRAY16KCLIENT is used in place of ECA_16KARRAYCLIENT, while variable name cannot start with a number.

---

message()

Returns the string representation of the status code

ADDFAIL = 168
Channel subscription request failed

ALLOCMEM = 48
Unable to allocate additional dynamic memory

BADCHID = 410
Invalid channel identifier

BADCOUNT = 176
Invalid element count requested

BADPRIORITY = 450
Invalid channel priority

BADSTR = 186
Invalid string

BADSYNCGRP = 354
Invalid synchronous group identifier

BADTYPE = 114
The data type specified is invalid

DBLCHNL = 200
Identical process variable names on multiple servers

DISCONN = 192
Virtual circuit disconnect

DISCONNCHID = 106
The request was ignored because the specified channel is disconnected

EVDISALLOW = 210
Request inappropriate within subscription (monitor) update callback

GETFAIL = 152
Channel read request failed

IODONE = 339
IO operations have completed

IOINPROGRESS = 347
IO operations are in progress

ISATTACHED = 424
Thread is already attached to a client context

NORMAL = 1
Normal successful completion

NOTTHREADED = 458
Preemptive callback not enabled - additional threads may not join context

NOWTACCESS = 376
Channel write request failed

PUTFAIL = 160
Channel write request failed

STRTOBIG = 96
The supplied string is unusually large

TIMEOUT = 80
User specified timeout on IO operation expired

TOLARGE = 72
The requested data transfer is greater than available memory or EPICS_CA_MAX_ARRAY_BYTES

UNAVAILINSERV = 432
Not supported by attached service

message()

Returns the string representation of the status code

severity()

Returns the severity of the status code

Return type CA_K

class caffi.ca.**DBF**
Enum redefined from DBF_XXX macros.

toSTS()
Returns DBR.STS_XXX

toTIME()
Returns DBR.TIME_XXX

toGR()
Returns DBR.GR_XXX

toCTRL()
Returns DBR.CTRL_XXX

CHAR = 4
uint8

DOUBLE = 6
double

ENUM = 3
uint16

FLOAT = 2
float

INT = 1
int16, synonym of SHORT

INVALID = -1
the channel's native type when disconnected

LONG = 5
int32

SHORT = 1
int16

STRING = 0
array of 40 characters

toCTRL()
Returns DBR.CTRL_XXX

toGR()
Returns DBR.GR_XXX

toSTS()
Returns DBR.STS_XXX

toTIME()
Returns DBR.TIME_XXX

class caffi.ca.**DBR**
Enum redefined from DBR_XXX macros.

isSTRING()
Returns True if type is STRING or one of XXX_STRING

isSHORT()

Returns True if type is SHORT or one of XXX_SHORT

isFLOAT()

Returns True if type is FLOAT or one of XXX_FLOAT

isENUM()

Returns True if type is ENUM or one of XXX_ENUM

isCHAR()

Returns True if type is CHAR or one of XXX_CHAR

isLONG()

Returns True if type is LONG or one of XXX_LONG

isDOUBLE()

Returns True if type is DOUBLE or one of XXX_DOUBLE

isPlain()

Returns True if type is one of STRING, SHORT, FLOAT, ENUM, CHAR, LONG, DOUBLE

isSTS()

Returns True if type is one of STS_XXX

isTIME()

Returns True if type is one of TIME_XXX

isGR()

Returns True if type is one of GR_XXX

isCTRL()

Returns True if type is one of CTRL_XXX

class caffi.ca.**DBE**

Enum redefined from DBE_XXX macros.

ALARM = 4

Trigger an event when the alarm state changes

ARCHIVE = 2

Trigger an event when an archive significant change in the channel's value occurs. Relies on the archiver monitor deadband field under DCT.

PROPERTY = 8

Trigger an event when a property change (control limit, graphical limit, status string, enum string ...) occurs.

VALUE = 1

Trigger an event when a significant change in the channel's value occurs. Relies on the monitor deadband field under DCT.

class caffi.ca.**ChannelState**

Enum redefined from C enum channel_state

CLOSED = 3

channel deleted

CONN = 2

valid chid, IOC was found, still available

NEVER_CONN = 0
valid chid, IOC not found

NEVER_SEARCH = 4
invalid chid

PREV_CONN = 1
valid chid, IOC was found, but unavailable

class caffi.ca.**CA_PRIORITY**
Enum redefined from CA_PRIORITY_XXX macros.

class caffi.ca.**CA_OP**
Enum redefined from C macros CA_OP_XXX

ADD_EVENT = 3
subscribe

CLEAR_EVENT = 4
unsubscribe

CONN_DOWN = 7
connection lost

CONN_UP = 6
connection established

CREATE_CHANNEL = 2
create channel

GET = 0
get value

OTHER = 5
other

PUT = 1
put value

class caffi.ca.**AlarmSeverity**
Enum redefined from C enum type epicsAlarmSeverity. Due to the enum difference between C and Python, the enum item name has been greatly simplified:

```
epicsSevNone  -> AlarmSeverity.No
epicsSevMinor -> AlarmSeverity.Minor
...
```

Note: *No* is used in place of *None*, which is Python keyword.

class caffi.ca.**AlarmCondition**
Enum redefined from C enum type epicsAlarmCondition. Due to the enum difference between C and Python, the enum item name has been greatly simplified:

```
epicsAlarmNone -> AlarmCondition.No
epicsAlarmRead -> AlarmCondition.Read
...
```

Note: *No* is used in place of *None*, which is Python keyword.

CHAPTER 3

Module caffi.dbr

caffi.dbr.**format_dbr**(*dbrType*, *count*, *dbrValue*, *use_numpy*)

Convert the specified dbr data structure to Python dict

Parameters

- **dbrType** – The data type, DBR_XXX
- **count** – The array element count
- **dbrValue** – A pointer of data of the specified type and number

Returns A dict filled with the values from the C structure fields.

CHAPTER 4

ChangeLog

4.1 1.0.2 (23-10-2017)

- Minor packaging changes.

4.2 1.0.1 (19-05-2017)

- Allow count=0 if ca.get is called with a valid callback function

4.3 1.0.0 (06-04-2017)

- Initial release.

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Python Module Index

C

Index