Channel Access Client Library Developers Guide

Jeff Hill

Testing Status for Success or Failure

- Nearly all functions return an integer status code
  - `ECA_NORMAL` is returned if successful
  - Unsuccessful status codes are listed with each routine in the manual
  - An error number and the error's severity are embedded in CA status (error) constants
    - Error numbers and severities can be converted to strings

Three Ways to Test Status

```
status = ca_XXXX();
SEVCHK ( status,
  "ca_XXXX() returned failure status");
if ( status & CA_M_SUCCESS ) {
  printf ( “It failed”);
}
if ( status != ECA_NORMAL ) {
  printf (
  “It failed because \"%s\"\n",
  ca_message ( status ) );
}
```

What is a CA Channel?

- A channel is a virtual communication link between a client application and a process variable
- Once created, a channel can be used to
  - Read PV's current value
  - Write PV's current value
  - Subscribe for PV state change notification

Creating a Channel

```
#include <cadef.h>
chid chan;
Int status = ca_create_channel
   ( “fred”, 0, 0, 0, & chan );
```

Channel Clean Up

```
#include <cadef.h>
int ca_clear_channel ( chan );
```

User Supplied Callback Functions

- Library employs callback based completion notification
- A C structure, `event_handler_args`, is typically passed to the application supplied callback
  - `status` field set to one of the CA client error codes
  - `dbr` field is a void pointer to any data that might be returned
    - Will be nill if status isn't ECA_NORMAL or data capsule isn't expected
  - `usr`, `chid`, and `type` are set to the values specified with the request

Asynchronously Returned Status

- Requests that appear to be valid to the client may fail in the server
  - Writing the string "off" to a floating point field is an example of this type of error
  - Typically, the CA client library function returns status indicating the validity of the request and whether it was successfully added to the server’s request queue
  - Communication of completion status is deferred until completion notify callback is called

Channel Access Exceptions

- Global exception notify handler
  - Server detects a failure, but there is no associated request, or the request does not have callback based completion notification
  - Certain internal exceptions within the CA client library
  - Failures detected by the SEVCHK macro
- The default global exception callback handler
  - Prints a message on the console
  - Exits if the exception condition is severe
  - May be replaced – see `ca_add_exception_event()`

Channel Access Data Types

- Arguments of type `chtype` specify user's data type
  - One of the `DBR_XXXX` codes from db_access.h
    - One for each of C's primitive types
    - Several more compound (C structured) types

Primitive Types

- One `DBR_XXXX` code for each C primitive type
- Data addresses are passed to library as type-less `void *` pointers
  - Use C `typedef dbr_xxxx_t` to define storage to be used with `DBR_XXXX` type code
  - Ensures that C data type is consistent with the `DBR_XXXX` type code
    - Proper initial interpretation of void pointer
    - Also portability and architecture independence

Compound (C Structured) Types

- Several compound (C structure) types include
  - Channel value
  - Additional process variable properties
    - units, limits, time stamp, or alarm status
  - See table in the CA reference manual, db_access.h

Thread Safety

- Starting with EPICS R3.14 the CA client library is thread safe on all OS
  - In past releases the library was thread safe only on `vxWorks`

Client Contexts

- Several CA client side tools running in the same address space (process) might need to be independent of each other
- For example, the database CA links and the sequencer are designed to not share the same CA client library threads, network circuits, and or data structures

Client Contexts

- Each thread that calls `ca_context_create()` for the first time, directly or implicitly, when calling a CA routine 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 threads spawned by the library are at fixed offsets from the priority of the thread that called `ca_context_create()`

Client Contexts

- User thread joins context
  - Call `ca_attach_context()`
  - Pass context identifier returned from `ca_current_context()` when it is called by thread that created the context
- A context can be cleaned up
  - Call `ca_context_destroy()`
  - First destroy any channels or application specific threads that are using it

Preemptive Callback to User Code

- Preemptive callback
  - User's callback functions might be called by CA's auxiliary threads when the user’s initiating thread is not executing in library
- Callbacks do not preempt callbacks
  - When the library invokes a user's callback it will wait for any current callback to complete prior to executing another callback function
- The programmer specifies if preemptive call back is enabled when creating a CA context
  - Preemptive call back is disabled by default.

Traditional Single Threaded Application

- To set up a traditional single threaded client you will need code like this

```
int status = ca_context_create (
  ca_disable_preemptive_callback );
SEVCHK ( status,
   “context create failed” );
```

- Application *must* periodically call ca_pend_event() to schedule library’s background activities

Preemptive Callback Based Application

- To set up a preemptive callback enabled CA client context you will need code like this

```
int status = ca_context_create (
  ca_enable_preemptive_callback );
SEVCHK ( status,
    “failed to create CA context” );
```

Connection Management

- CA servers will be restarted, and that network connectivity is transient
- New channels are typically initially in a disconnected state
- The library continuously monitors the network and tries to keep channels in a connected state

Synchronizing With Channel Connection State Changes

- Block in `ca_pend_io()` after creating channel(s)
  - Simple, short lifespan applications don't work well if connectivity changes
  - Nill connection state notify callback function pointer must be supplied
- Install connection state callback notification when creating the channel
  - Long Lifespan, connection state driven applications
  - `ca_pend_io ()` will not block

Native Type and Native Element Count

- When connected, a channels storage type and maximum element count are cached in the client library
  - `ca_field_type ( chid )`
  - `ca_element_count ( chid )`

Request Queuing

- Requests to a CA server are queued
- This queue is not flushed until…
  - One of ca_flush_io, ca_pend_io, ca_pend_event, or ca_sg_pend are called
  - Maximum queue size is exceeded
- Several requests may be combined
  - Important efficiently gains when passing through OS and network layers

Read Process Variable

- Ordinary get
  - Directly updates your variable
  - Simple applications
  - Value returned must not be used until success is returned from ca_pend_io()
- Get callback
  - Value is returned to completion notify callback

Ordinary Get Example

```
#include <cadef.h>
dbr_double_t val;
int status = ca_get (
  DBR_DOUBLE, chan, &val  );
```

Get Callback Example

```
#include <cadef.h>
void myCallback (
   struct event_handler_args ){}

int status = ca_get_callback (
   DBR_DOUBLE, chan,
   myCallback, 0 );
```

Write Process Variable

- Ordinary put
  - No response message if successful
    - Optimized network usage
    - Failure notification goes to global exception callback handler
  - If record is busy, intermediate values are dropped
- Put callback
  - Response message always sent
    - After cascaded record processing completes
  - If record is busy, intermediate values are *not* dropped

Ordinary Put Example

```
#include <cadef.h>

dbr_double_t val = 3.3;
int status = ca_put (
   DBR_DOUBLE, chan, & val );
```

Put Callback Example

```
#include <cadef.h>
void myCallback (
  struct event_handler_args ){}
dbr_double_t val = 3.3;
int status = ca_put_callback (
  DBR_DOUBLE, chan, & val,
  myCallback, 0 );
```

Subscribe For Process Variable State Change Updates

- Only one request message, but many response messages
  - Optimized network usage
  - Lower latency state change notification
- Currently three update triggering events
  - GUI value state change
  - Archive value state change
  - Alarm status and severity state change

Subscription Install Example

```
#include <cadef.h>
void myCallBack (
  struct event_handler_args ) {}
unsigned long nElements = 1;
evid id;
int status = ca_create_subscription
 ( DBR_DOUBLE, nElements, chan,
  DBE_VALUE, myCallback, 0, & id );
```

Subscription Cancel Example

```
#include <cadef.h>

int status =
  ca_clear_subscription (
   id );
```

Interface Pitfalls

- Not periodically calling ca_pend_event() in single threaded application
  - Circuits backup to server
    - IP kernel buffer shortage in IOC
  - Channels don't connect or disconnect unexpectedly
- The purposes of ca_pend_event() and ca_pend_io() are frequent source of confusion
  - `ca_pend_event()` schedules background activities
  - `ca_pend_io()` blocks for completion

Load Inducing Pitfalls

- Continuously creating / destroying channels
  - Very heavy load on network / servers
    - TCP circuit startup / tear down
    - Broadcast traffic
- Neglecting to destroy the channel
  - Leaks resources allocated in the server