EPICS R3.14 Channel Access Reference Manual

Jeffrey O. Hill

Los Alamos National Laboratory, SNS Division

Ralph Lange

Helmholtz-Zentrum Berlin (BESSY II)

Modified on $Date: 2009/07/30 23:09:54 $

Table of Contents

Troubleshooting

Function Call Interface Guidelines

Functionality Index

Function Call Interface Index

Deprecated Function Call Interface Function Index

Return Codes

Configuration

Why Reconfigure Channel Access

Typically reasons to reconfigure EPICS Channel Access:

- Two independent control systems must share a network without fear of interaction
- A test system must not interact with an operational system
- Use of address lists instead of broadcasts for name resolution and server beacons
- Control system occupies multiple IP subnets
- Nonstandard client disconnect time outs or server beacon intervals

- Specify the local time zone
- Transport of large arrays

EPICS Environment Variables

All Channel Access (CA) configuration occurs through EPICS environment variables. When searching for an EPICS environment variable EPICS first looks in the environment using the ANSI C getenv() call. If no matching variable exists then the default specified in the EPICS build system configuration files is used.

Name	Range	Default
EPICS_CA_ADDR_LIST	{N.N.N.N N.N.N.N:P ...}	<none>
EPICS_CA_AUTO_ADDR_LIST	{YES, NO}	YES
EPICS_CA_CONN_TMO	r > 0.1 seconds	30.0
EPICS_CA_BEACON_PERIOD	r > 0.1 seconds	15.0
EPICS_CA_REPEATER_PORT	i > 5000	5065
EPICS_CA_SERVER_PORT	i > 5000	5064
EPICS_CA_MAX_ARRAY_BYTES	i >= 16384	16384
EPICS_CA_MAX_SEARCH_PERIOD	r > 60 seconds	300
EPICS_TS_MIN_WEST	-720 < i <720 minutes	360

Environment variables are set differently depending on the command line shell that is in use.

C shell	setenv EPICS_CA_ADDR_LIST 1.2.3.4
bash	export EPICS_CA_ADDR_LIST=1.2.3.4
vxWorks shell	putenv ("EPICS_CA_ADDR_LIST =1.2.3.4")
DOS command line	set EPICS_CA_ADDR_LIST=1.2.3.4
Windows NT / 2000 / XP	control panel / system / environment tab

CA and Wide Area Networks

Normally in a local area network (LAN) environment CA discovers the address of the host for an EPICS process variable by broadcasting frames containing a list of channel names (CA search messages) and waiting for responses from the servers that host the channels identified. Likewise CA clients efficiently discover that CA servers have recently joined the LAN or disconnected from the LAN by monitoring periodically broadcasted beacons sent out by the servers. Since hardware broadcasting requires special hardware capabilities, we are required to provide additional configuration information when EPICS is extended to operate over a wide area network (WAN).

IP Network Administration Background Information

Channel Access is implemented using internet protocols (IP). IP addresses are divided into host and network portions. The boundary between each portion is determined by the IP netmask. Portions of the IP address corresponding to zeros in the netmask specify the hosts address within an IP subnet. Portions of the IP address corresponding to binary ones in the netmask specify the address of a host's IP subnet. Normally the scope of a broadcasted frame will be limited to one IP subnet. Addresses with the host address portion set to all zeros or all ones are special. Modern IP kernel implementations reserve destination addresses with the host portion set to all ones for the purpose of addressing broadcasts to a particular subnet. In theory we can issue a broadcast frame on any broadcast capable LAN within the interconnected internet by specifying the proper subnet address combined with a host portion set to all ones. In practice these "directed broadcasts" are frequently limited by the default router configuration. The proper directed broadcast address required to reach a particular host can be obtained by logging into that host and typing the command required by your local operating environment. Ignore the loop back interface and use the broadcast address associated with an interface connected to a path through the network to your client. Typically there will be only one Ethernet interface.

UNIX	ifconfig -a
vxWorks	ifShow
Windows	ipconfig

IP ports are positive integers. The IP address, port number, and protocol type uniquely identify the source and destination of a particular frame transmitted between computers. Servers are typically addressed by a well known port number. Clients are assigned a unique ephemeral port number during initialization. IP ports below 1024 are reserved for servers that provide standardized facilities such as mail or file transfer. Port number between 1024 and 5000 are typically reserved for ephemeral port number assignments.

IP port numbers

The two default IP port numbers used by Channel Access may be reconfigured. This might occur when a site decides to set up two or more completely independent control systems that will share the same network. For instance, a site might set up an operational control system and a test control system on the same network. In this situation it is desirable for the test system and the operational system to use identical PV names without fear of collision. A site might also configure the CA port numbers because some other facility is already using the default port numbers. The default Channel Access port numbers have been registered with IANA.

Purpose	Default	Environment Variable
CA Server	5064	EPICS_CA_SERVER_PORT
CA Beacons (sent to CA repeater daemon)	5065	EPICS_CA_REPEATER_PORT

If a client needs to communicate with two servers that are residing at different port numbers then an extended syntax may be used with the EPICS_CA_ADDR_LIST environment variable. See WAN Environment below.

WAN Environment

When the CA client library connects a channel it must first determine the IP address of the server the channels Process Variable resides on. To accomplish this the client sends name resolution (search) requests to a list of server destination addresses. These server destination addresses can be IP unicast addresses (individual host addresses) or IP broadcast addresses. Each name resolution (search) request contains a list of Process Variable names.If one of the servers reachable by this address list knows the IP address of a CA server that can service one or more of the specified Process Variables, then it sends back a response containing the server's IP address and port number.

During initialization CA builds the list of server destination addresses used when sending CA client name resolution (search) requests. This table is initialized by introspecting the network interfaces attached to the host. For each interface found that is attached to a broadcast capable IP subnet, the broadcast address of that subnet is added to the list. For each point to point interface found, the destination address of that link is added to the list. This automatic server address list initialization can be disabled if the EPICS environment variable "EPICS_CA_AUTO_ADDR_LIST" exists and its value is either of "no" or "NO". The typical default is to enable network interface introspection driven initialization with "EPICS_CA_AUTO_ADDR_LIST" set to "YES" or "yes".

Following network interface introspection, any IP addresses specified in the EPICS environment variable EPICS_CA_ADDR_LIST are added to the list of destination addresses for CA client name resolution requests. In an EPICS system crossing multiple subnets the EPICS_CA_ADDR_LIST must be set so that CA name resolution (search requests) frames pass from CA clients to the targeted CA servers unless a CA proxy (gateway) is installed. The addresses in EPICS_CA_ADDR_LIST may be dotted IP addresses or host names if the local OS has support for host name to IP address translation. When multiple names are added to EPICS_CA_ADDR_LIST they must be separated by white space. There is no requirement that the addresses specified in the EPICS_CA_ADDR_LIST be a broadcast addresses, but this will often be the most convenient choice.

C shell	setenv EPICS_CA_ADDR_LIST "1.2.3.255 8.9.10.255"
bash	export EPICS_CA_ADDR_LIST="1.2.3.255 8.9.10.255"
vxWorks	putenv ("EPICS_CA_ADDR_LIST=1.2.3.255 8.9.10.255")

If a client needs to communicate with two servers that are residing at different port numbers then an extended syntax may be used with the EPICS_CA_ADDR_LIST environment variable. Each host name or IP address in the EPICS_CA_ADDR_LIST may be immediately followed by a colon and an IP port number without intervening whitespace. Entries that do not specify a port number will default to EPICS_CA_SERVER_PORT.

C shell	setenv EPICS_CA_ADDR_LIST "1.2.3.255 8.9.10.255:10000"

Routing Restrictions on vxWorks Systems

Frequently vxWorks systems boot by default with routes limiting access only to the local subnet. If a EPICS system is operating in a WAN environment it may be necessary to configure routes into the vxWorks system which enable a vxWorks based CA server to respond to requests originating outside it's subnet. These routing restrictions can also apply to vxWorks base CA clients communicating with off subnet servers. An EPICS system manager can implement an rudimentary, but robust, form of access control for a particular host by not providing routes in that host that reach outside of a limited set of subnets. See "routeLib" in the vxWorks reference manual.

Disconnect Time Out Interval

If the CA client library does not see a beacon from a server that it is connected to for EPICS_CA_CONN_TMO seconds then an state-of-health message is sent to the server over TCP/IP. If this state-of-health message isn't promptly replied to then the client library will conclude that channels communicating with the server are no longer responsive and inform the CA client side application via function callbacks. The parameter EPICS_CA_CONN_TMO is specified in floating point seconds. The default is typically 30 seconds. For efficient operation it is recommended that EPICS_CA_CONN_TMO be set to no less than twice the value specified for EPICS_CA_BEACON_PERIOD.

Prior to EPICS R3.14.5 an unresponsive server implied an immediate TCP circuit disconnect, immediate resumption of UDP based search requests, and immediate attempts to reconnect. There was concern about excessive levels of additional activity when servers are operated close to the edge of resource limitations. Therefore with version R3.14.5 and greater the CA client library continues to inform client side applications when channels are unresponsive, but does not immediately disconnect the TCP circuit. Instead the CA client library postpones circuit shutdown until receiving indication of circuit disconnect from the IP kernel. This can occur either because a server is restarted or because the IP kernel's internal TCP circuit inactivity keep alive timer has expired after a typically long duration (as is appropriate for IP based systems that need to avoid thrashing during periods of excessive load). The net result is less search and TCP circuit setup and shutdown activity suring periods of excessive load.

Dynamic Changes in the CA Client Library Search Interval

The CA client library will continuously attempt to connect any CA channels that an application has created until it is successful. The library periodically queries the server destination address list described above with name resolution requests for any unresolved channels. Since this address list frequently contains broadcast addresses, and because nonexistent process variable names are frequently configured, or servers may be temporarily unavailable, then it is necessary for the CA client library internals to carefully schedule these requests in time to avoid introducing excessive load on the network and the servers.

When the CA client library has many channels to connect, and most of its name resolution requests are responded to, then it sends name resolution requests at an interval that is twice the estimated round trip interval for the set of servers responding, or at the minimum delay quantum for the operating system - whichever is greater. The number of UDP frames per interval is also dynamically adjusted based on the past success rates.

If a name resolution request is not responded to, then the client library doubles the delay between name resolution attempts and reduces the number of requests per interval. The maximum delay between attempts is limited by EPICS_CA_MAX_SEARCH_PERIOD (see Configuring the Maximum Search Period). Note however that prior to R3.14.7, if the client library did not receive any responses over a long interval it stoped sending name resolution attempts altogether until a beacon anomaly was detected (see below).

The CA client library continually estimates the beacon period of all server beacons received. If a particular server's beacon period becomes significantly shorter or longer then the client is said to detect a beacon anomaly. The library boosts the search interval for unresolved channels when a beacon anomaly is seen or when *any* successful search response is received, but with a longer initial interval between requests than is used when the application creates a channel. Creation of a new channel does *not* (starting with EPICS R3.14.7) change the interval used when searching for preexisting unresolved channels. The program "casw" prints a message on standard out for each CA client beacon anomaly detect event.

See also When a Client Does not See the Server's Beacon.

Configuring the Maximum Search Period

The rate at which name resolution (search) requests are sent exponentially backs off to a plateau rate. The value of this plateau has an impact on network traffic because it determines the rate that clients search for channel names that are miss-spelled or otherwise don't exist in a server. Furthermore, for clients that are unable to see the beacon from a new server, the plateau rate may also determine the maximum interval that the client will wait until discovering a new server.

Starting with EPICS R3.14.7 this maximum search rate interval plateau in seconds is determined by the EPICS_CA_MAX_SEARCH_PERIOD environment variable.

See also When a Client Does not See the Server's Beacon.

The CA Repeater

When several client processes run on the same host it is not possible for all of them to directly receive a copy of the server beacon messages when the beacon messages are sent to unicast addresses, or when legacy IP kernels are still in use. To avoid confusion over these restrictions a special UDP server, the CA Repeater, is automatically spawned by the CA client library when it is not found to be running. This program listens for server beacons sent to the UDP port specified in the EPICS_CA_REPEATER_PORT parameter and fans any beacons received out to any CA client program running on the same host that have registered themselves with the CA Repeater. If the CA Repeater is not already running on a workstation, then the "caRepeater" program must be in your path before using the CA client library for the first time.

If a host based IOC is run on the same workstation with standalone CA client processes, then it is probably best to start the caRepeater process when the workstation is booted. Otherwise it is possible for the standalone CA client processes to become dependent on a CA repeater started within the confines of the host based IOC. As long as the host based IOC continues to run there is nothing wrong with this situation, but problems could arise if this host based IOC process exits before the standalone client processes which are relying on its CA repeater for services exit.

Since the repeater is intended to be shared by multiple clients then it could be argued that it makes less sense to set up a CA repeater that listens for beacons on only a subset of available network interfaces. In the worst case situation the client library might see beacon anomalies from servers that it is not interested in. Modifications to the CA repeater forcing it to listen only on a subset of network interfaces might be considered for a future release if there appear to be situations that require it.

Configuring the Time Zone

Note: Starting with EPICS R3.14 all of the libraries in the EPICS base distribution rely on facilities built into the operating system to determine the correct time zone. Nevertheless, several programs commonly used with EPICS still use the original "tssubr" library and therefore they still rely on proper configuration of EPICS_TS_MIN_WEST.

While the CA client library does not translate inbetween the local time and the time zone independent internal storage of EPICS time stamps, many EPICS client side applications call core EPICS libraries which provide these services. To set the correct time zone users must compute the number of positive minutes west of GMT (maximum 720 inclusive) or the negative number of minutes east of GMT (minimum -720 inclusive). This integer value is then placed in the variable EPICS_TS_MIN_WEST.

Time Zone	EPICS_TS_MIN_WEST
USA Eastern	300
USA Central	360
USA Mountain	420
USA Pacific	480
Alaska	540
Hawaii	600
Japan	-540
China	-420
Germany	-120
United Kingdom	0

Configuring the Maximum Array Size

Starting with version R3.14 the environment variable EPICS_CA_MAX_ARRAY_BYTES determines the size of the largest array that may pass through CA. Prior to this version only arrays smaller than 16k bytes could be transfered. The CA libraries maintains a free list of 16384 byte network buffers that are used for ordinary communication. If EPICS_CA_MAX_ARRAY_BYTES is larger than 16384 then a second free list of larger data buffers is established and used only after a client send its first large array request.

The CA client library uses EPICS_CA_MAX_ARRAY_BYTES to determines the maximum array that it will send or receive. Likewise, the CA server uses EPICS_CA_MAX_ARRAY_BYTES to determine the maximum array that it may send or receive. The client does not influence the server's message size quotas and visa versa. In fact the value of EPICS_CA_MAX_ARRAY_BYTES need not be the same in the client and the server. If the server receives a request which is too large to read or respond to in entirety then it sends an exception message to the client. Likewise, if the CA client library receives a request to send an array larger than EPICS_CA_MAX_ARRAY_BYTES it will return ECA_TOLARGE.

A common mistake is to correctly calculate the maximum datum size in bytes by multiplying the number of elements by the size of a single element, but neglect to add additional bytes for the compound data types (for example DBR_GR_DOUBLE) commonly used by the more sophisticated client side applications. *Based on this confusion, one could arrive at the conclusion that EPICS_CA_MAX_ARRAY_BYTES might have been better named EPICS_CA_MAX_DATUM_BYTES, or that the software should be changed internally to round the users request up by the size of the maximum scalar datum (nothing has been done to address this issue so far).*

Configuring a CA Server

Name	Range	Default
EPICS_CAS_SERVER_PORT	i > 5000	EPICS_CA_SERVER_PORT
EPICS_CAS_AUTO_BEACON_ADDR_LIST	{YES, NO}	EPICS_CA_AUTO_ADDR_LIST
EPICS_CAS_BEACON_ADDR_LIST	{N.N.N.NN.N.N.N:P...}	EPICS_CA_ADDR_LIST[1]
EPICS_CAS_BEACON_PERIOD	r > 0.1 seconds	EPICS_CA_BEACON_PERIOD
EPICS_CAS_BEACON_PORT	i > 5000	EPICS_CA_REPEATER_PORT
EPICS_CAS_INTF_ADDR_LIST	{N.N.N.NN.N.N.N:P...}	<none>
EPICS_CAS_IGNORE_ADDR_LIST	{N.N.N.NN.N.N.N:P...}	<none>

Server Port

The server configures its port number from the EPICS_CAS_SERVER_PORT environment variable if it is specified. Otherwise the EPICS_CA_SERVER_PORT environment variable determines the server's port number. Two servers can share the same UDP port number on the same machine, but there are restrictions - see a discussion of unicast addresses and two servers sharing the same UDP port on the same host.

Server Beacons

The EPICS_CAS_BEACON_PERIOD parameter determines the server's beacon period and is specified in floating point seconds. The default is typically 15 seconds. See also EPICS_CA_CONN_TMO and Dynamic Changes in the CA Client Library Search Interval.

CA servers build a list of addresses to send beacons to during initialization. If EPICS_CAS_AUTO_BEACON_ADDR_LIST has the value "YES" then the beacon address list will be automatically configured to contain the broadcast addresses of all LAN interfaces found in the host and the destination address of all point-to-point interfaces found in the host. However, if the user also defines EPICS_CAS_INTF_ADDR_LIST then beacon address list automatic configuration is constrained to the network interfaces specified therein, and therefore only the broadcast addresses of the specified LAN interfaces, and the destination addresses of all specified point-to-point links, will be automatically configured.

If EPICS_CAS_BEACON_ADDR_LIST is defined then its contents will be used to augment any automatic configuration of the beacon address list. Individual entries in EPICS_CAS_BEACON_ADDR_LIST may override the destination port number if ":nnn" follows the host name or IP address there. Alternatively, when both EPICS_CAS_BEACON_ADDR_LIST and EPICS_CAS_INTF_ADDR_LIST are not defined then the

contents of EPICS_CA_ADDR_LIST is used to augment the list. Otherwise, the list is not augmented.

The EPICS_CAS_BEACON_PORT parameter specifies the destination port for server beacons. The only exception to this occurs when ports are specified in EPICS_CAS_BEACON_ADDR_LIST or possibly in EPICS_CA_ADDR_LIST. If EPICS_CAS_BEACON_PORT is not specified then beacons are sent to the port specified in EPICS_CA_REPEATER_PORT.

Binding a Server to a Limited Set of Network Interfaces

The parameter EPICS_CAS_INTF_ADDR_LIST allows a ca server to bind itself to, and therefore accept messages only over, a limited set of the local host's network interfaces (each specified by it's IP address). On UNIX systems type "netstat -i" (type "ipconfig" on windows) to see a list of the local host's network interfaces. Specifically, UDP search messages addressed to both the IP addresses in EPICS_CAS_INTF_ADDR_LIST and also to the broadcast addresses of the corresponding LAN interfaces will be accepted by the server. By default, the CA server is accessible from all network interfaces configured into its host. *In R3.14 and previous releases the CA server employed by iocCore does not implemet this feature*.

Ignoring Process Variable Name Resolution Requests From Certain Hosts

Name resolution requests originating from any of the IP addresses specified in the EPICS_CAS_IGNORE_ADDR_LIST parameter are not replied to.*In R3.14 and previous releases the CA server employed by iocCore does not implemet this feature*.

Client Configuration that also Applies to Servers

See also Configuring the Maximum Array Size.

See also Routing Restrictions on vxWorks Systems.

Building an Application

Required Header (.h) Files

An application that uses the CA client library functions described in this document will need to include the cadef.h header files as follows.

```
#include "cadef.h"
```

This header file is located at "<EPICS base>/include/". It includes many other header files (operating system specific and otherwise), and therefore the application must also specify "<EPICS base>/include/os/<arch>" in its header file search path.

Required Libraries

An application that uses the Channel Access Client Library functions described in this document will need to link with the EPICS CA Client Library and also the EPICS Common Library. The EPICS CA Client Library calls the EPICS Common Library. The following table shows the names of these libraries on UNIX and Windows systems.

	UNIX Object	**UNIX Shareable**	**Windows Object**	**Windows Shareable**
EPICS CA Client Library	libca.a	libca.so	ca.lib	ca.dll
EPICS Common Library	libCom.a	libCom.so	Com.lib	Com.dll

The above libraries are located in "<EPICS base>/lib/<architechture>".

Compiler and System Specific Build Options

If you do not use the EPICS build environemnt (layered make files) then it may be helpful to run one of the EPICS make files and watch the compile/link lines. This may be the simplest way to capture the latest system and compiler specific options required by your build environment. I have included some snapshots of typical build lines below, but expect some risk of this information becoming dated.

Typical Linux Build Options

```
/usr/bin/gcc -c -D_POSIX_C_SOURCE=199506L -D_POSIX_THREADS -D_XOPEN_SOURCE=500 -
DOSITHREAD_USE_DEFAULT_STACK -D_X86_ -DUNIX -D_BSD_SOURCE -Dlinux -D_REENTRANT -ansi -O3
-Wall -I. -I.. -I../../../include/os/Linux -I../../../include ../acctst.c

/usr/bin/g++ -o acctst -L/home/user/epicsR3.14/epics/base/lib/linux-x86/ -Wl,-
rpath,/mnt/bogart_home/hill/epicsR3.14/epics/base/lib/linux-x86 acctstMain.o acctst.o -
lca -lCom
```

Typical Solaris Build Options

```
/opt/SUNWspro/bin/cc -c -D_POSIX_C_SOURCE=199506L -D_XOPEN_SOURCE=500 -
DOSITHREAD_USE_DEFAULT_STACK -DUNIX -DSOLARIS=9 -mt -D__EXTENSIONS__ -Xc -v -xO4 -I. -
I.. -I./../../../include/os/solaris -I./../../../include ../acctst.c

/opt/SUNWspro/bin/CC -o acctst -L/home/phoebus1/JHILL/epics/base/lib/solaris-sparc/ -mt
-z ignore -z combreloc -z lazyload -R/home/disk1/user/epics/base/lib/solaris-sparc
acctstMain.o acctst.o -lca -lCom
```

Typical Windows Build Options

```
cl -c /nologo /D__STDC__=0 /Ox /GL /W3 /w44355 /MD -I. -I.. -
I..\\..\\..\\include\\os\\WIN32 -I..\\..\\..\\include ..\\acctst.c

link -nologo /LTCG /incremental:no /opt:ref /release /version:3.14 -out:acctst.exe
acctstMain.obj acctst.obj d:/user/R3.14.clean/epics/base/lib/WIN32-x86/ca.lib
d:/user/R3.14.clean/epics/base/lib/WIN32-x86/
```

Typical vxWorks Build Options

```
/usr/local/xcomp/ppc/bin/ccppc -c -D_POSIX_SOURCE -DCPU=PPC603 -DvxWorks -include
/home/vx/tornado20/target/h/vxWorks.h -ansi -O3 -Wall -mcpu=603 -mstrict-align -fno-
builtin -I. -I.. -I../../../include/os/vxWorks -I../../../include -
I/home/vx/tornado20/target/h ../acctst.c
```

Other Systems and Compilers

Contributions gratefully accepted.

Command Line Utilities

acctst

```
acctst <PV name> [progress logging level] [channel duplication count]
                 [test repetition count] [enable preemptive callback]
```

Description

Channel Access Client Library regression test.

The PV used with the test must be native type DBR_DOUBLE or DBR_FLOAT, and modified only by acctst while the test is running. Therefore, periodically scanned hardware attached analog input records do not work well. Test failure is indicated if the program stops prior to printing "test complete". If unspecified the progress logging level is zero, and no messages are printed while the test is progressing. If unspecified, the channel duplication count is 20000. If unspecified, the test repetition count is once only. If unspecified, preemptive callback is disabled.

catime

```
catime <PV name> [channel count] [append number to pv name if true]
```

Description

Channel Access Client Library performance test.

If unspecified, the channel count is 10000. If the "append number to pv name if true" argument is specified and it is greater than zero then the channel names in the test are numbered as follows.

<PV name>000000, <PV name>000001, ... <PV name>nnnnnn

casw

```
casw [-i <interest level>]
```

Description

CA server "beacon anomaly" logging.

CA server beacon anomalies occur when a new server joins the network, a server is rebooted, network connectivity to a server is reestablished, or if a server's CPU exits a CPU load saturated state.

CA clients with unresolved channels reset their search request scheduling timers whenever they see a beacon anomaly.

This program can be used to detect situations where there are too many beacon anomalies. IP routing configuration problems may result in false beacon anomalies that might cause CA clients to use unnecessary additional network bandwidth and server CPU load when searching for unresolved channels.

If there are no new CA servers appearing on the network, and network connectivity remains constant, then casw should print no messages at all. At higher interest levels the program prints a message for every beacon that is received, and anomalous entries are flagged with a star.

caEventRate

```
caEventRate <PV name> [subscription count]
```

Description

Connect to the specified PV, subscribe for monitor updates the specified number of times (default once), and periodically log the current sampled event rate, average event rate, and the standard deviation of the event rate in Hertz to standard out.

ca_test

```
ca_test <PV name> [value to be written]
```

Description

If a value is specified it is written to the PV. Next, the current value of the PV is converted to each of the many external data type that can be specified at the CA client library interface, and each of these is formated and then output to the console.

Command Line Tools

caget

```
caget [options] <PV name> ...
```

Description

Get and print value for PV(s).

The values for one or multiple PVs are read and printed to stdout. The DBR_... format in which the data is read, the output format, and a number of details of how integer and float values are represented can be

controlled using command line options.

When getting multiple PVs, their order on the command line is retained in the output.

<table>
<tr><th>Option</th><th>Description</th></tr>
<tr><td>-h</td><td>Print usage information</td></tr>
<tr><td></td><td>CA options:</td></tr>
<tr><td>-w <sec></td><td>Wait time, specifies CA timeout, default is 1.0 second(s)</td></tr>
<tr><td>-c</td><td>Asynchronous get (use ca_get_callback and wait for completion)</td></tr>
<tr><td>-p <prio></td><td>CA priority (0-99, default 0=lowest)</td></tr>
<tr><td></td><td>Format and data type options:</td></tr>
<tr><td></td><td>Default output format is "name value"</td></tr>
<tr><td>-t</td><td>Terse mode - print only value, without name</td></tr>
<tr><td>-a</td><td>Wide mode "name timestamp value stat sevr" (read PVs as DBR_TIME_xxx)</td></tr>
<tr><td>-n</td><td>Print DBF_ENUM values as number (default are enum strings)</td></tr>
<tr><td>-d <type></td><td>Request specific dbr type; use string (DBR_ prefix may be omitted)

or number of one of the following types:

<table>
<tr><td>DBR_STRING</td><td>0</td><td>DBR_STS_FLOAT</td><td>9</td><td>DBR_TIME_LONG</td><td>19</td><td>DBR_CTRL_SHORT</td><td>29</td></tr>
<tr><td>DBR_INT</td><td>1</td><td>DBR_STS_ENUM</td><td>10</td><td>DBR_TIME_DOUBLE</td><td>20</td><td>DBR_CTRL_INT</td><td>29</td></tr>
<tr><td>DBR_SHORT</td><td>1</td><td>DBR_STS_CHAR</td><td>11</td><td>DBR_GR_STRING</td><td>21</td><td>DBR_CTRL_FLOAT</td><td>30</td></tr>
<tr><td>DBR_FLOAT</td><td>2</td><td>DBR_STS_LONG</td><td>12</td><td>DBR_GR_SHORT</td><td>22</td><td>DBR_CTRL_ENUM</td><td>31</td></tr>
<tr><td>DBR_ENUM</td><td>3</td><td>DBR_STS_DOUBLE</td><td>13</td><td>DBR_GR_INT</td><td>22</td><td>DBR_CTRL_CHAR</td><td>32</td></tr>
<tr><td>DBR_CHAR</td><td>4</td><td>DBR_TIME_STRING</td><td>14</td><td>DBR_GR_FLOAT</td><td>23</td><td>DBR_CTRL_LONG</td><td>33</td></tr>
<tr><td>DBR_LONG</td><td>5</td><td>DBR_TIME_INT</td><td>15</td><td>DBR_GR_ENUM</td><td>24</td><td>DBR_CTRL_DOUBLE</td><td>34</td></tr>
<tr><td>DBR_DOUBLE</td><td>6</td><td>DBR_TIME_SHORT</td><td>15</td><td>DBR_GR_CHAR</td><td>25</td><td>DBR_STSACK_STRING</td><td>37</td></tr>
<tr><td>DBR_STS_STRING</td><td>7</td><td>DBR_TIME_FLOAT</td><td>16</td><td>DBR_GR_LONG</td><td>26</td><td>DBR_CLASS_NAME</td><td>38</td></tr>
<tr><td>DBR_STS_SHORT</td><td>8</td><td>DBR_TIME_ENUM</td><td>17</td><td>DBR_GR_DOUBLE</td><td>27</td><td></td><td></td></tr>
<tr><td>DBR_STS_INT</td><td>8</td><td>DBR_TIME_CHAR</td><td>18</td><td>DBR_CTRL_STRING</td><td>28</td><td></td><td></td></tr>
</table>
</td></tr>
<tr><td></td><td>Arrays:</td></tr>
<tr><td></td><td>Value format: Print number of requested values, then list of values</td></tr>
<tr><td>Default:</td><td>Print all values</td></tr>
<tr><td>-# <count></td><td>Print first <count> elements of an array</td></tr>
<tr><td>-S</td><td>Print array of char as a string (long string)</td></tr>
<tr><td></td><td>Floating point type format:</td></tr>
<tr><td>Default:</td><td>Use %g format</td></tr>
<tr><td>-e <nr></td><td>Use %e format, with <nr> digits after the decimal point</td></tr>
<tr><td>-f <nr></td><td>Use %f format, with <nr> digits after the decimal point</td></tr>
<tr><td>-g <nr></td><td>Use %g format, with <nr> digits after the decimal point</td></tr>
<tr><td>-s</td><td>Get value as string (may honour server-side precision)</td></tr>
<tr><td></td><td>Integer number format:</td></tr>
<tr><td>Default:</td><td>Print as decimal number</td></tr>
<tr><td>-0x</td><td>Print as hex number</td></tr>
<tr><td>-0o</td><td>Print as octal number</td></tr>
<tr><td>-0b</td><td>Print as binary number</td></tr>
</table>

camonitor

```
camonitor [options] <PV name> ...
```

Description

Subscribe to and print value updates for PV(s).

Option	Description
-h	Print usage information
	CA options:
-w <sec>	Wait time, specifies CA timeout, default is 1.0 second(s)
-m <mask>	Specify CA event mask to use, with <mask> being any combination of 'v' (value), 'a' (alarm), 'l' (log). Default: va
-p <prio>	CA priority (0-99, default 0=lowest)
	Timestamps:
Default:	Print absolute timestamps (as reported by CA server)
-t <key>	Specify timestamp source(s) and type, with <key> containing 's' = CA server (remote) timestamps 'c' = CA client (local) timestamps (shown in '()'s) 'n' = no timestamps 'r' = relative timestamps (time elapsed since start of program) 'i' = incremental timestamps (time elapsed since last update) 'I' = incremental timestamps (time elapsed since last update, by channel)
	Enum Format:
-n	Print DBF_ENUM values as number (default are enum strings)
	Arrays:
	Value format: Print number of requested values, then list of values
Default:	Print all values
-# <count>	Print first <count> elements of an array
-S	Print array of char as a string (long string)
	Floating point type format:
Default:	Use %g format
-e <nr>	Use %e format, with <nr> digits after the decimal point
-f <nr>	Use %f format, with <nr> digits after the decimal point
-g <nr>	Use %g format, with <nr> digits after the decimal point
-s	Get value as string (may honour server-side precision)
	Integer number format:
Default:	Print as decimal number
-0x	Print as hex number
-0o	Print as octal number
-0b	Print as binary number

caput

```
caput [options] <PV name> <value>
caput -a [options] <PV name> <no of elements> <value> ...
```

Description

Put value to a PV.

The specified value is written to the PV (as a string). The PV value is read before and after the write operation and printed as "Old" and "new" values on stdout.

The array variant writes an array to the specified PV. The first numeric argument specifying the number of

array elements is kept for compatibility with the array data format of caget - the actual number of values specified on the command line is used.

Option	Description
-h	Print usage information
	CA options:
-w <sec>	Wait time, specifies CA timeout, default is 1.0 second(s)
-c	Asynchronous put (use ca_put_callback and wait for completion)
-p <prio>	CA priority (0-99, default 0=lowest)
	Format options:
-t	Terse mode - print only sucessfully written value, without name
	Enum Format:
	Auto - try value as ENUM string, then as index number
-n	Force interpretation of values as numbers
-s	Force interpretation of values as strings
	Arrays:
-a	Put array data
	Value format: Print number of requested values, then list of values
-S	Put string as an array of char (long string)

cainfo

```
cainfo [options] <PV name> ...
```

Description

Get and print channel and connection information for PV(s).

All available Channel Access related information about PV(s) is printed to stdout.

The -s option allows to specify an interest level for calling Channel Access' internal report function ca_client_status(), that prints lots of internal informations on stdout, including environment settings, used CA ports etc.

Option	Description
-h	Print usage information
	CA options:
-w <sec>	Wait time, specifies CA timeout, default is 1.0 second(s)
-s <level>	Call ca_client_status with the specified interest level
-p <prio>	CA priority (0-99, default 0=lowest)

excas

excas [options]

This is an example CA server that is sometimes used for testing purposes. An example server can be created with the makeBaseApp perl script, as descibed in the application Developer's Guide.

Option	Description
-d <uuuu>	set level uuuu for debug messages, where uuuu is an positive integer number
-p <aaaa>	prefix all of the PV names below with aaaa changing, for example, the name of "bill" to "xyz:bill"
-t <n.n>	set execution time where n.n is a positive real number

-c <uuuu>	set the numbered alias count
-s <nnn>	the default, nnn is one, enables periodic scanning of the PV replacing the PV with its value added with a small random change, when nnn is zero it turns off this type of periodic scanning
-ss <nnn>	the default, nnn is one, enables synchronous scanning, and if nnn is zero it turns on asynchronous scanning
-ad <n.n>	set the delay before asynchronous operations complete (defaults to 0.1 seconds)
-an <nnn>	set the maximum number of simultaneous asynchronous operations (defaults to 1000)

The example server has a compile time fixed set of example variables.

Process Variable Name	Number of Elements	IO Type	Data Type	High Limit	Low Limit	Scan Period
jane	1	Synchronous	float point, 64 bits	10.0	0.0	0.1 Seconds, random noise changes
fred	1	Synchronous	float point, 64 bits	10.0	-10.0	2.0 Seconds, random noise changes
janet	1	Asynchronous	float point, 64 bits	10.0	0.0	0.1 Seconds, random noise changes
freddy	1	Asynchronous	float point, 64 bits	10.0	-10.0	2.0 Seconds, random noise changes
alan	100	Synchronous	float point, 64 bits	10.0	-10.0	2.0 Seconds, random noise changes
albert	1000	Synchronous	float point, 64 bits	10.0	-10.0	20.0 Seconds, random noise changes
boot	1	Synchronous	enumerated, 16 bits	10.0	-10.0	changed only by client
booty	1	Asynchronous	enumerated, 16 bits	10.0	-10.0	1.0 Seconds, random noise changes
bill	1	Synchronous	float point, 64 bits	10.0	-10.0	changed only by client
billy	1	Asynchronous	float point, 64 bits	10.0	-10.0	changed only by client
bloaty	100000	Synchronous	float point, 64 bits	10.0	-10.0	changed only by client

Bugs

Not all of the options listed above have been tested recently.

Troubleshooting

When Clients Do Not Connect to Their Server

Client and Server Broadcast Addresses Dont Match

Verify that the broadcast addresses are identical on the server's host and on the client's host. This can be checked on UNIX with "netstat -i" or "ifconfig -a"; on vxWorks with ifShow; and on windows with ipconfig. It is normal for the broadcast addresses to not be identical if the client and server are not directly attached to the same IP subnet, and in this situation the EPICS_CA_ADDR_LIST must be set. Otherwise, if the client and server are intended to be on the same IP subnet, then the problem may be that the IP netmask is incorrectly set in the network interface configuration. On most operating systems, when the host's IP address is configured, the host's IP subnet mask is also configured.

Client Isn't Configured to Use the Server's Port

Verify that the client and server are using the same UDP port. Check the server's port by running "netstat -a | grep nnn" where nnn is the port number configured in the client. If you do not set EPICS_CA_SERVER_PORT or EPICS_CAS_SERVER_PORT then the default port will be 5064.

Unicast Addresses in the EPICS_CA_ADDR_LIST Does not Reliably Contact Servers Sharing the Same UDP Port on the Same Host

Two servers can run on the same host with the same server port number, but there are restrictions. If the host has a modern IP kernel it is possible to have two or more servers share the same UDP port. It is not possible for these servers to run on the same host using the same TCP port. If the CA server library detects that a server is attempting to start on the same port as an existing CA server then both servers will use the same UDP port, and the 2nd server will be allocated an ephemeral TCP port. Clients can be configured to use the same port number for both servers. They will locate the 2nd server via the shared UDP port, and transparently connect to the 2nd server's ephemeral TCP port. Be aware however that If there are two server's running on the same host sharing the same UDP port then they will both receive UDP search requests sent as broadcasts, but unfortunately (due to a weakness of most IP kernel implementations) only one of the servers will typically receive UDP search requests sent to unicast addresses (i.e. a single specific host's ip address).

Client Does not See Server's Beacons

Two conclusions deserve special emphasis. *First, if a client does not see the server's beacons, then it will use additional network and server resources sending periodic state-of-health messages. Second, if a client does not see a newly introduced server's beacon, then it will take up to EPICS_CA_MAX_SEARCH_PERIOD to find that newly introduced server*. Also, starting with EPICS R3.14.7 the client library does *not* suspend searching for a channel after 100 unsuccessful attempts until a beacon anomaly is seen. Therefore, if the client library is from before version R3.14.7 of EPICS and it timed out attempting to find a server whoose beacon cant be seen by the client library then the client application might need to be restarted in order to connect to this new beacon-out-of-range server. The typical situation where a client would not see the server's beacon might be when the client isnt on the same IP subnet as the server, and the client's EPICS_CA_ADDR_LIST was modified to include a destination address for the server, but the server's beacon address list was not modified so that it's beacons are received by the client.

A Server's IP Address Was Changed

When communication over a virtual circuit times out, then each channel attached to the circuit enters a disconnected state and the disconnect callback handler specified for the channel is called. However, the circuit is not disconnected until TCP/IP's internal, typically long duration, keep alive timer expires. The disconnected channels remain attached to the beleaguered circuit and no attempt is made to search for, or to reestablish, a new circuit. If, at some time in the future, the circuit becomes responsive again, then the attached channels enter a connected state again and reconnect call back handlers are called. Any monitor subscriptions that received an update message while the channel was disconnected are also refreshed. If at any time the library receives an indication from the operating system that a beleaguered circuit has shutdown or was disconnected then the library will immediately reattempt to find servers for each channel and connect circuits to them.

A well known negative side effect of the above behavior is that CA clients will wait the full (typically long) duration of TCP/IP's internal keep alive timer prior to reconnecting under the following scenario (all of the following occur):

- An server's (IOC's) operating system crashes (or is abruptly turned off) or a vxWorks system is stopped by any means
- This operating system does not immediately reboot using the same IP address
- A duplicate of the server (IOC) is started appearing at a different IP address

It is unlikely that any rational organization will advocate the above scenario in a production system. Nevertheless, there *are* opportunities for users to become confused during control system *development*, but it is felt that the robustness improvements justify isolated confusion during the system integration and checkout activities where the above scenarios are most likely to occur.

Contrast the above behavior with the CA client library behavior of releases prior to R3.14.5 where the beleaguered circuit was immediately closed when communication over it timed out. Any attached channels were immediately searched for, and after successful search responses arrived then attempts were made to

build a new circuit. This behavior could result in undesirable resource consumption resulting from periodic circuit setup and teardown overhead (thrashing) during periods of CPU / network / IP kernel buffer congestion.

Put Requests Just Prior to Process Termination Appear to be Ignored

Short lived CA client applications that issue a CA put request and then immediately exit the process (return from `main` or call `exit`) may find that there request isn't executed. To guarantee that the request is sent call `ca_flush` followed by `ca_context_destroy` prior to terminating the process.

ENOBUFS Messages

Many Berkley UNIX derived Internet Protocol (IP) kernels use a memory management scheme with a fixed sized low level memory allocation quantum called an "mbuf". Messages about "ENOBUFS" are an indication that your IP kernel is running low on mbuf buffers. An IP kernel mbuf starvation situation may lead to temporary IP communications stalls or reduced throughput. This issue has to date been primarily associated with vxWorks systems where mbuf starvation on earlier vxWorks versions is rumored to lead to permanent IP communications stalls which are resolved only by a system reboot. IP kernels that use mbufs frequently allow the initial and maximum number of mbufs to be configured. Consult your OS's documentation for configuration procedures which vary between OS and even between different versions of the same OS.

Contributing Circumstances

- The total number of connected clients is high. Each active socket requires dedicated mbufs for protocol control blocks, and for any data that might be pending in the operating system for transmission to Channel Access or to the network at a given instant. If you increase the vxWorks limit on the maximum number of file descriptors then it may also be necessary to increase the size of the mbuf pool.

- The server has multiple connections where the server's sustained event (monitor subscription update) production rate is higher than the client's or the network's sustained event consumption rate. This ties up a per socket quota of mbufs for data that are pending transmission to the client via the network. In particular, if there are multiple clients that subscribe for monitor events but do not call ca_pend_event() or ca_poll() to process their CA input queue, then a significant mbuf consuming backlog can occur in the server.

- The server does not get a chance to run (because some other higher priority thread is running) and the CA clients are sending a high volume of data over TCP or UDP. This ties up a quota of mbufs for each socket in the server that isn't being reduced by the server's socket read system calls.

- The server has multiple stale connections. Stale connections occur when a client is abruptly turned off or disconnected from the network, and an internal "keepalive" timer has not yet expired for the virtual circuit in the operating system, and therefore mbufs may be dedicated to unused virtual circuits. This situation is made worse if there are active monitor subscriptions associated with stale connections which will rapidly increase the number of dedicated mbufs to the quota available for each circuit.
- When sites switch to the vxWorks 5.4 IP kernel they frequently run into network pool exhaustion problems. This may be because the original vxWorks IP kernel expanded the network pool as needed at runtime while the new kernel's pool is statically configured at compile time, and does *not* expand as needed at runtime. Also, at certain sites problems related to vxWorks network driver pool exhaustion have also been reported (this can also result in ENOBUF diagnostic messages).

Related Diagnostics

- The EPICS command "casr [interest level]" displays information about the CA server and how many clients are connected.
- The vxWorks command "inetstatShow" indicates how many bytes are pending in mbufs and indirectly (based on the number of circuits listed) how many mbuf based protocol control blocks have been consumed. The vxWorks commands (availability depending on vxWorks version) mbufShow, netStackSysPoolShow, and netStackDataPoolShow indicate how much space remains in the network stack pool.
- The RTEMS command "netstat [interest level]" displays network information including mbuf consumption statistics.

Server Subscription Update Queuing

If the subscription update producer in the server produces subscription updates faster than the subscription update consumer in the client consumes them, then events have to be discarded if the buffering in the server isn't allowed to grow to an infinite size. This is a law of nature – based on queuing theory of course.

What is done depends on the version of the CA server. All server versions place quotas on the maximum number of subscription updates allowed on the subscription update queue at any given time. If this limit is reached, an intervening update is discarded in favor of a more recent update. Depending on the version of the server, rapidly updating subscriptions are or are not allowed to cannibalize the quotas of slow updating subscriptions in limited ways. Nevertheless, there is always room on the queue for at least one update for each subscription. This guarantees that the most recent update is always sent.

Adding further complication, the CA client library also implements a primitive type of flow control. If the client library sees that it is reading a large number of messages one after another w/o intervening delay it knows that it is not consuming events as fast as they are produced. In that situation it sends a message telling the server to temporarily stop sending subscription update messages. When the client catches up it sends another message asking the server to resume with subscription updates. This prevents slow clients from getting time warped, but also guarantees that intervening events are discarded until the slow client catches up.

There is currently no message on the IOC's console when a particular client is slow on the uptake. A message of this type used to exist many years ago, but it was a source of confusion (and what we will call message noise) so it was removed.

There is unfortunately no field in the protocol allowing the server to indicate that an intervening subscription update was discarded. We should probably add that capability in a future version. Such a feature would, for example, be beneficial when tuning an archiver installation.

Function Call Interface General Guidelines

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_pend are called allowing several operations to be efficiently sent over the network together. Any process variable values written into your program's variables by ca_get() should not be referenced by your program until ECA_NORMAL has been received from ca_pend_io().

Status Codes

If successful, the routines described here return the status code ECA_NORMAL. Unsuccessful status codes returned from the client library are listed with each routine in this manual. 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 ca_xxx() 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. An error number and the error's severity are embedded in CA status (error) constants. Applications shouldn't test the success of a CA function call by checking to see if the returned value is zero as is the UNIX convention. Below are several methods to test CA function returns. See ca_signal() and SEVCHK for more information on this topic.

```
status = ca_XXXX();
SEVCHK( status, "ca_XXXX() returned failure status");

if ( status & CA_M_SUCCESS ) {
        printf ( "The requested ca_XXXX() operation didn't complete successfully");
}

if ( status != ECA_NORMAL ) {
        printf("The requested ca_XXXX() operation didn't complete successfully because \"%s\"\n",
                ca_message ( status ) );
}
```

Channel Access Data Types

CA channels form a virtual circuit between a process variable (PV) and a client side application program. It is possible to connect a wide variety of data sources into EPICS using the CA server library. When a CA channel communicates with an EPICS Input Output Controller (IOC) then a field is a specialization of a PV, and an EPICS record is a plug compatible function block that contains fields, and the meta data below frequently are mapped onto specific fields within the EPICS records by the EPICS record support (see the EPICS Application Developer Guide).

Arguments of type chtype specifying the data type you wish to transfer. They expect one of the set of DBR_XXXX data type codes defined in db_access.h. There are data types for all of the C primitive types, and there are also compound (C structure) types that include various process variable properties such as units, limits, time stamp, or alarm status. The primitive C types follow a naming convention where the C typedef dbr_xxxx_t corresponds to the DBR_XXXX data type code. The compound (C structure) types follow a naming convention where the C structure tag dbr_xxxx corresponds to the DBR_XXXX data type code. The following tables provides more details on the structure of the CA data type space. Since data addresses are passed to the CA client library as typeless "void *" pointers then care should be taken to ensure that you have passed the correct C data type corresponding to the DBR_XXXX type that you have specified. Architecture independent types are provided in db_access.h to assist programmers in writing portable code. For example "dbr_short_t" should be used to send or receive type DBR_SHORT. Be aware that type name DBR_INT has been deprecated in favor of the less confusing type name DBR_SHORT. In practice, both the DBR_INT type code and the DBR_SHORT type code refer to a 16 bit integer type, and are functionally equivalent.

Channel Access Primitive Data Types

CA Type Code	Primitive C Data Type	Data Size
DBR_CHAR	dbr_char_t	8 bit character
DBR_SHORT	dbr_short_t	16 bit integer
DBR_ENUM	dbr_enum_t	16 bit unsigned integer
DBR_LONG	dbr_long_t	32 bit signed integer
DBR_FLOAT	dbr_float_t	32 bit IEEE floating point
DBR_DOUBLE	dbr_double_t	64 bit IEEE floating point
DBR_STRING	dbr_string_t	40 character string

Structure of the Channel Access Data Type Space

CA Type Code	Read / Write	Primitive C Data Type	Process Variable Properties
DBR_<PRIMITIVE TYPE>	RW	dbr_<primitive type>_t	value
DBR_STS_<PRIMITIVE TYPE>	R	struct dbr_sts_<primitive type>	value, alarm status, and alarm severity
DBR_TIME_<PRIMITIVE TYPE>	R	struct dbr_time_<primitive type>	value, alarm status, alarm severity, and time stamp
DBR_GR_<PRIMITIVE TYPE>	R	struct dbr_gr_<primitive type>	value, alarm status, alarm severity, units, display precision, and graphic limits
DBR_CTRL_<PRIMITIVE TYPE>	R	struct dbr_ctrl_<primitive type>	value, alarm status, alarm severity, units, display precision, graphic limits, and control limits
DBR_PUT_ACKT	W	dbr_put_ackt_t	Used for global alarm acknowledgement. Do transient alarms have to be acknowledged? (0,1) means (no, yes).
DBR_PUT_ACKS	W	dbr_put_acks_t	Used for global alarm acknowledgement. The highest alarm severity to acknowledge. If the current alarm severity is less then or equal to this value the alarm is acknowledged.
		struct	

DBR_STSACK_STRING	R	struct dbr_stsack_string	value, alarm status, alarm severity, ackt, ackv
DBR_CLASS_NAME	R	dbr_class_name_t	name of enclosing interface (name of the record if channel is attached to EPICS run time database)

Channel value arrays can also be included within the structured CA data types. If more than one element is requested, then the individual elements can be accessed in an application program by indexing a pointer to the value field in the DBR_XXX structure. For example, the following code computes the sum of the elements in a array process variable and prints its time stamp. The dbr_size_n function can be used to determine the correct number of bytes to reserve when there are more than one value field elements in a structured CA data type.

```
#include <stdio.h>
#include <stdlib.h>

#include "cadef.h"

int main ( int argc, char ** argv )
{
    struct dbr_time_double * pTD;
    const dbr_double_t * pValue;
    unsigned nBytes;
    unsigned elementCount;
    char timeString[32];
    unsigned i;
    chid chan;
    double sum;
    int status;

    if ( argc != 2 ) {
        fprintf ( stderr, "usage: %s <channel name>", argv[0] );
        return -1;
    }

    status = ca_create_channel ( argv[1], 0, 0, 0, & chan );
    SEVCHK ( status, "ca_create_channel()" );
    status = ca_pend_io ( 15.0 );
    if ( status != ECA_NORMAL ) {
        fprintf ( stderr, "\"%s\" not found.\n", argv[1] );
        return -1;
    }

    elementCount = ca_element_count ( chan );
    nBytes = dbr_size_n ( DBR_TIME_DOUBLE, elementCount );
    pTD = ( struct dbr_time_double * ) malloc ( nBytes );
    if ( ! pTD ) {
        fprintf ( stderr, "insufficient memory to complete request\n" );
        return -1;
    }

    status = ca_array_get ( DBR_TIME_DOUBLE, elementCount, chan, pTD );
    SEVCHK ( status, "ca_array_get()" );
    status = ca_pend_io ( 15.0 );
    if ( status != ECA_NORMAL ) {
        fprintf ( stderr, "\"%s\" didnt return a value.\n", argv[1] );
        return -1;
    }

    pValue = & pTD->value;
    sum = 0.0;
    for ( i = 0; i < elementCount; i++ ) {
        sum += pValue[i];
    }

    epicsTimeToStrftime ( timeString, sizeof ( timeString ),
        "%a %b %d %Y %H:%M:%S.%f", & pTD->stamp );

    printf ( "The sum of elements in %s at %s was %f\n",
        argv[1], timeString, sum );

    ca_clear_channel ( chan );
    ca_task_exit ();
    free ( pTD );
```

```
    return 0;
}
```

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_callback, ca_get_callback, and ca_add_event all request notification of asynchronous completion via this mechanism. The `event_handler_args` structure is passed *by value* to the application supplied callback. In this structure the `dbr` field is a void pointer to any data that might be returned. The `status` field will be set to one of the CA error codes in caerr.h and will indicate the status of the operation performed in the IOC. If the status field isn't set to ECA_NORMAL or data isn't normally returned from the operation (i.e. put call back) then you should expect that the `dbr` field will be set to a nill pointer (zero). The fields `usr`, `chid`, and `type` are set to the values specified when the request was made by the application. The "dbr" pointer, and any data that it points to, are valid only when executing within the user's callback function.

```
typedef struct event_handler_args {
    void            *usr;   /* user argument supplied with request */
    chanId          chid;   /* channel id */
    long            type;   /* the type of the item returned */
    long            count;  /* the element count of the item returned */
    const void      *dbr;   /* a pointer to the item returned */
    int             status; /* ECA_XXX status of the requested op from the server */
} evargs;

void myCallback ( struct event_handler_args args )
{
    if ( args.status != ECA_NORMAL ) {
    }
    if ( args.type == DBR_TIME_DOUBLE ) {
         const struct dbr_time_double * pTD =
              ( const struct dbr_time_double * ) args.dbr;
    }
}
```

Channel Access Exceptions

When the server detects a failure, and there is no client call back function attached to the request, then 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().

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

If the Process Variable's server and it's client are colocated within the same memory address space and the same host then the ca_xxx() operations bypass the server and directly interact with the server tool component (commonly the IOC's function block database). In this situation the ca_xxx() routines 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.

Arrays

For routines 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 less 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. Starting with EPICS R3.14 the maximum array size may be configured in the client and in the server.

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 a nill connection handler 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).

Thread Safety and Preemptive Callback to User Code

Starting with EPICS R3.14 the CA client libraries are fully thread safe on all OS (in past releases the library was thread safe only on vxWorks). When the client library is initialized the programmer may specify if preemptive call back is enabled. Preemptive call back is disabled by default. If preemptive call back is enabled then the user's call back 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 call back 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 call back function it will always wait for the current callback to complete prior to executing another call back function. Programmers enabling preemptive callback should be familiar with using mutex locks to create a reliable multi-threaded program.

To set up a traditional single threaded client you will need code like this (see ca_context_create and CA Client Contexts and Application Specific Auxiliary Threads) .

```
SEVCHK ( ca_context_create(ca_disable_preemptive_callback ), "application pdq calling
ca_context_create" );
```

To set up a preemptive callback enabled CA client context you will need code like this (see ca_context_create and CA Client Contexts and Application Specific Auxiliary Threads).

```
SEVCHK ( ca_context_create(ca_enable_preemptive_callback ), "application pdq calling
ca_context_create" );
```

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_context_create() 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_context_create(). 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 ca_context_create(ca_enable_preemptive_callback). It is not possible to attach a thread to a non-preemptive CA context created explicitly *or implicitly* with ca_create_context(ca_disable_preemptive_callback). Once a thread has joined with a CA context it need only make ordinary ca_xxxx() library 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_context_destroy(). The context may be created and destroyed by different threads as long as they are both part of the same context.

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. In single threaded applications a file descriptor manager like Xt or the interface described in fdManager.h can be used to monitor both mouse clicks and also CA's file descriptors so that ca_poll() can be called immediately when CA server messages arrives over the network.

Avoid Emulating Bad Practices that May Still be Common

With the embryonic releases of EPICS it was a common practice to examine a channel's connection state, its native type, and its native element count by directly accessing fields in a structure using a pointer stored in type `chid`. Likewise, a user private pointer in the per channel structure was also commonly set by directly accessing fields in the channel structure. A number of difficulties arise from this practice, which has long since been deprecated. For example, prior to release 3.13 it was recognized that transient changes in certain private fields in the per channel structure would make it difficult to reliably test the channels connection state using these private fields directly. Therefore, in release 3.13 the names of certain fields were changed to discourage this practice. Starting with release 3.14 codes written this way will not compile. Codes intending to maintain the highest degree of portability over a wide range of EPICS versions should be especially careful. For example you should replace all instances off `channel_id->count` with `ca_element_count(channel_id)`. This approach should be reliable on all versions of EPICS in use today. The construct `ca_puser(chid) = xxxx` is particularly problematic. The best mechanisms for setting the per channel private pointer will be to pass the user private pointer in when creating the channel. This approach is implemented on all versions. Otherwise, you can also use `ca_set_puser(CHID,PUSER)`, but this function is available only after the first official (post beta) release of EPICS 3.13.

Calling CA Functions from the vxWorks Shell Thread

Calling CA functions from the vxWorks shell thread is a somewhat questionable practice for the following reasons.

- The vxWorks shell thread runs at the very highest priority in the system and therefore socket calls are made at a priority that is above the priority of tNetTask – a practice that has caused the WRS IP kernel to get sick in the past. That symptom was observed some time ago, but we don't know if WRS has fixed the problem.
- The vxWorks shell thread runs at the very highest priority in the system and therefore certain CA auxiliary threads will not get the priorities that are requested for them. This might cause problems only when in a CPU saturation situations.
- If the code does not call ca_context_destroy (ca_task_exit in past releases) then resources are left dangling.
- In EPICS R3.13 the CA client library installed vxWorks task exit handlers behaved strangely if CA functions were called from the vxWorks shell, ca_task_exit() wasn't called, and the vxWorks shell restarted. In EPICS R3.14 vxWorks task exit handlers are not installed and therefore cleanup is solely the responsibility of the user. With EPICS R3.14 the user must call ca_context_destroy or ca_task_exit to clean up on vxWorks. This is the same behavior as on all other OS.

Calling CA Functions from POSIX signal handlers

As you might expect, it isnt safe to call the CA client library from a POSIX signal handler. Likewise, it isnt safe to call the CA client library from interrupt context.

Function Call Reference

`ca_context_create()`

```
#include <cadef.h>
enum ca_preemptive_callback_select
    { ca_disable_preemptive_callback, ca_enable_preemptive_callback };
int ca_context_create ( enum ca_preemptive_callback_select SELECT );
```

Description

This function, or ca_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 `ca_disable_preemptive_callback` is specified then additional threads are *not* allowed to join the CA context using ca_context_attach() because allowing other threads to join implies that CA callbacks will be called preemptively from more than one thread.

Arguments

`SELECT`
: This argument specifies if preemptive invocation of callback functions is allowed. If so your callback functions might be called when the thread that calls this routine is not executing in the CA client library. There are two implications to consider.

  First, if preemptive callback mode is enabled the developer must provide mutual exclusion protection for his data structures. In this mode it's possible for two threads to touch the application's data structures at once: this might be the initializing thread (the thread that called ca_context_create) and also a private thread created by the CA client library for the purpose of receiving network messages and calling callbacks. It might be prudent for developers who are unfamiliar with mutual exclusion locking in a multi-threaded environment to specify `ca_disable_preemptive_callback`.

  Second, if preemptive callback mode is enabled the application is no longer burdened with the necessity of periodically polling the CA client library in order that it might take care of its background activities. If `ca_enable_preemptive_callback` is specified then CA client background activities, such as connection management, will proceed even if the thread that calls this routine is not executing in the CA client library. Furthermore, in preemptive callback mode callbacks might be called with less latency because the library is not required to wait until the initializing thread (the thread that called ca_context_create) is executing within the CA client library.

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)

See Also

ca_context_destroy()

`ca_context_destroy()`

```
#include <cadef.h>
void ca_context_destroy();
```

Description

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. A program running in an IOC context must delete all of its channels prior to calling ca_context_destroy() to avoid a crash.

A CA client application that calls epicsExit() *must* install an EPICS exit handler that calls ca_context_destroy() only *after* first calling ca_create_context(). This will guarantee that the EPICS exit handlers get called in the correct order.

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 ca_context_destroy() hasn't been called, but on light weight systems such as vxWorks or RTEMS no

cleanup occurs unless the application call ca_context_destroy().

Returns

ECA_NORMAL - Normal successful completion

See Also

ca_context_create()

ca_create_channel()

```
#include <cadef.h>
typedef void ( *pCallBack ) (
         struct connection_handler_args );
int ca_create_channel
(
        const char     *PROCESS_VARIABLE_NAME,
        caCh           *USERFUNC,
        void           *PUSER,
        capri          priority,
        chid           *PCHID
);
```

Description

This function creates a CA channel. 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 ca_clear_channel() is used to release these resources. If successful, the routine writes a channel identifier into the user's variable of type "chid". 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 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 specifying a nil connection call back function pointer.
- The second method requires that you register a connection handler by supplying a valid connection callback function pointer. This connection handler is called whenever the connection state of the channel changes. If you have installed a connection handler then ca_pend_io() will *not* block waiting for the channel to enter a connected state.

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

Due to the inherently transient nature of network connections the order of connection call backs relative to the order that ca_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.

Example

See caExample.c in the example application created by makeBaseApp.pl.

Arguments

`PROCESS_VARIABLE_NAME`
: A nil terminated process variable name string. EPICS process control function block database variable names are of the form "<record name>.<field name>". If the field name and the period separator are omitted then the "VAL" field is implicit. For example "RFHV01" and "RFHV01.VAL" reference the

same EPICS process control function block database variable.

USERFUNC
Optional address of the user's call back function to be run when the connection state changes. Casual users of channel access may decide to set this field to nil or 0 if they do not need to have a call back function run in response to each connection state change event.

The following structure is passed *by value* to the user's connection connection callback function. The op field will be set by the CA client library to CA_OP_CONN_UP when the channel connects, and to CA_OP_CONN_DOWN when the channel disconnects. See ca_puser if the PUSER argument is required in your callback handler.

```
struct  ca_connection_handler_args {
    chanId  chid;  /* channel id */
    long    op;    /* one of CA_OP_CONN_UP or CA_OP_CONN_DOWN */
};
```

PUSER
The value of this void pointer argument is retained in storage associated with the specified channel. See the MACROS manual page for reading and writing this field. Casual users of channel access may wish to set this field to nil or 0.

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.

PCHID
The user supplied channel identifier storage is overwritten with a channel identifier if this routine is successful.

Returns

ECA_NORMAL - Normal successful completion

ECA_BADTYPE - Invalid DBR_XXXX type

ECA_STRTOBIG - Unusually large string

ECA_ALLOCMEM - Unable to allocate memory

ca_clear_channel()

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

Description

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

All remote operation requests such as the above are accumulated (buffered) and not forwarded to the IOC until one of ca_flush_io, ca_pend_io, ca_pend_event, or ca_sg_pend 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.

Arguments

CHID
Identifies the channel to delete.

Returns

ECA_NORMAL - Normal successful completion

ECA_BADCHID - Corrupted CHID

`ca_put()`

```
#include <cadef.h>
int ca_put ( chtype TYPE,
        chid CHID, void *PVALUE );
int ca_array_put ( chtype TYPE,
        unsigned long COUNT,
        chid CHID, const void *PVALUE);
typedef void ( *pCallBack ) (struct event_handler_args );
int ca_put_callback ( chtype TYPE,
        chid CHID, const void *PVALUE,
        pCallBack PFUNC, void *USERARG );
int ca_array_put_callback ( chtype TYPE,
        unsigned long COUNT,
        chid CHID, const void *PVALUE,
        pCallBack PFUNC, void *USERARG );
```

Description

Write a scalar or array value to a process variable.

When ca_array_put or ca_put are invoked 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 ca_array_put_callback are invoked 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 ca_flush_io, ca_pend_io, ca_pend_event, or ca_sg_pend are called. This allows several requests to be efficiently combined into one message.

Description (IOC Database Specific)

A CA put request causes the record to process if the record's SCAN field is set to passive, and the field being written has it's process passive attribute set to true. If such a record is already processing when a put request is initiated the specified field is written immediately, and the record is scheduled to process again as soon as it finishes processing. Earlier instances of multiple put requests initiated while the record is being processing may be discarded, but the last put request initiated is always written and processed.

A CA put *callback* request causes the record to process if the record's SCAN field is set to passive, and the field being written has it's process passive attribute set to true. For such a record, the user's put callback function is not called until after the record, and any records that the record links to, finish processing. If such a record is already processing when a put *callback* request is initiated the put *callback* request is postponed until the record, and any records it links to, finish processing.

If the record's SCAN field is not set to passive, or the field being written has it's process passive attribute set to false then the CA put or CA put callback request cause the specified field to be immediately written, but they do not cause the record to be processed.

Arguments

TYPE
: The external type of the supplied value to be written. Conversion will occur if this does not match the native type. Specify one from the set of DBR_XXXX in db_access.h

COUNT
: Element count to be written to the specified channel. This must match the array pointed to by PVALUE.

CHID
: Channel identifier

PVALUE
: Pointer to a value or array of values provided by the application to be written to the channel.

PFUNC
: address of user supplied callback function to be run when the requested operation completes

USERARG
: pointer sized variable retained and then passed back to user supplied function above

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

See Also

ca_flush_io()

ca_pend_event()

ca_sg_array_put()

ca_get()

```
#include <cadef.h>
int ca_get ( chtype TYPE,
        chid CHID, void *PVALUE );
int ca_array_get ( chtype TYPE, unsigned long COUNT,
        chid CHID, void *PVALUE );
typedef void ( *pCallBack ) (struct event_handler_args );
int ca_get_callback ( chtype TYPE,
        chid CHID, pCallBack USERFUNC, void *USERARG);
int ca_array_get_callback ( chtype TYPE, unsigned long COUNT,
        chid CHID,
        pCallBack USERFUNC, void *USERARG );
```

Description

Read a scalar or array value from a process variable.

When ca_get or ca_array_get are invoked the returned channel value cant be assumed to be stable in the application supplied buffer until after ECA_NORMAL is returned from ca_pend_io. If a connection is lost outstanding ca get requests are not automatically reissued following reconnect.

When ca_get_callback or ca_array_get_callback are invoked a value is read from the channel and then the user's callback is invoked with a pointer to the retrieved value. Note that ca_pend_io will not block for the delivery of values requested by ca_get_callback. 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 ca_flush_io, ca_pend_io, ca_pend_event, or ca_sg_pend are called. This allows several requests to be efficiently sent over the network in one message.

Description (IOC Database Specific)

A CA get or CA get callback request causes the record's field to be read immediately independent of whether the record is currently being processed or not. There is currently no mechanism in place to cause a record to be processed when a CA get request is initiated.

Example

See caExample.c in the example application created by makeBaseApp.pl.

Arguments

`TYPE`
: The external type of the user variable to return the value into. Conversion will occur if this does not match the native type. Specify one from the set of DBR_XXXX in db_access.h

`COUNT`
: Element count to be read from the specified channel. Must match the array pointed to by PVALUE.

`CHID`
: Channel identifier

`PVALUE`
: Pointer to an application supplied buffer where the current value of the channel is to be written.

`USERFUNC`
: Address of user supplied callback function to be run when the requested operation completes.

`USERARG`
: Pointer sized variable retained and then passed back to user supplied call back function above.

Returns

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

See Also

ca_pend_io()

ca_pend_event()

ca_sg_array_get()

ca_create_subscription()

```
#include <cadef.h>
typedef void ( *pCallBack ) (
        struct event_handler_args );
int ca_create_subscription ( chtype TYPE,
        unsigned long COUNT, chid CHID,
        unsigned long MASK, pCallBack USERFUNC, void *USERARG,
        evid *PEVID );
```

Description

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

Subscriptions may be installed or canceled against both connected and disconnected channels. The specified USERFUNC 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 USERFUNC is called immediately after establishing a connection (or reconnection) with the process variable. The specified USERFUNC is called immediately with the process variable's current state from within ca_add_event() 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 ca_flush_io, ca_pend_io, ca_pend_event, or ca_sg_pend 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.

A better name for this function might have been ca_subscribe.

Example

See caMonitor.c in the example application created by makeBaseApp.pl.

Arguments

`TYPE`
: The type of value presented to the call back funstion. Conversion will occur if it does not match native type. Specify one from the set of DBR_XXXX in db_access.h

`COUNT`
: The element count to be read from the specified channel. A count of zero specifies the native elemnt count.

`CHID`
: channel identifier

`USRERFUNC`
: The address of user supplied callback function to be invoked with each subscription update.

`USERARG`
: pointer sized variable retained and passed back to user callback function

RESERVED
Reserved for future use. Specify 0.0 to remain upwardly compatible.

PEVID
This is a pointer to user supplied event id which is overwritten if successful. This event id can later be used to clear a specific event. This option may may be omitted by passing a nil pointer.

MASK
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 the following constants.

- DBE_VALUE - Trigger events when the channel value exceeds the monitor dead band
- DBE_ARCHIVE (or DBE_LOG) - Trigger events when the channel value exceeds the archival dead band
- DBE_ALARM - Trigger events when the channel alarm state changes
- DBE_PROPERTY - Trigger events when a channel property changes.

For functions above that do not include a trigger specification, events will be triggered when there are significant changes in the channel's value or when there are changes in the channel's alarm state. This is the same as "DBE_VALUE | DBE_ALARM."

Returns

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

See Also

ca_pend_event()

ca_flush_io()

ca_clear_subscription()

```
#include <cadef.h>
int ca_clear_subscription ( evid EVID );
```

Description

Cancel a subscription.

All ca_clear_event() requests such as the above are accumulated (buffered) and not forwarded to the server until one of ca_flush_io, ca_pend_io, ca_pend_event, or ca_sg_pend are called. This allows several requests to be efficiently sent together in one message.

Arguments

EVID
event id returned by ca_add_event()

Returns

ECA_NORMAL - Normal successful completion

ECA_BADCHID - Corrupted CHID SEE ALSO ca_add_event()

ca_pend_io()

```
#include <cadef.h>
int ca_pend_io ( double TIMEOUT );
```

Description

This function flushes the send buffer and then blocks until outstanding ca_get requests complete, and until channels created specifying nill connection handler function pointers connect for the first time.

- If ECA_NORMAL is returned then it can be safely assumed that all outstanding ca_get requests have completed successfully and channels created specifying nill connection handler function pointers have connected for the first time.
- If ECA_TIMEOUT is returned then it must be assumed for all previous ca_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 ca_pend_io(). Specifically, the function will block only for outstanding ca_get requests issued, and also any channels created specifying a nill connection handler function pointer, after the last call to ca_pend_io() or ca client context creation whichever is later. Note that ca_create_channel requests generally should not be reissued for the same process variable unless ca_clear_channel is called first.

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

The delay specified to ca_pend_io() 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 ca_pend_event, this routine will not process CA's background activities if none of the selected IO requests are pending.

Arguments

TIMEOUT
: Specifies the time out interval. A TIMEOUT interval of zero specifies forever.

Returns

ECA_NORMAL - Normal successful completion

ECA_TIMEOUT - Selected IO requests didnt complete before specified timeout

ECA_EVDISALLOW - Function inappropriate for use within an event handler

See Also

ca_get()

ca_create_channel()

ca_test_io()

ca_test_io()

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

Description

This function tests to see if all ca_get requests are complete and channels created specifying a nill connection callback function pointer are connected. It will report the status of outstanding ca_get requests issued, and channels created specifying a nill connection callback function pointer, 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

See Also

ca_pend_io()

ca_pend_event()

```
#include <cadef.h>
int ca_pend_event ( double TIMEOUT );
int ca_poll ();
```

Description

When ca_pend_event is invoked the send buffer is flushed and CA background activity is processed for TIMEOUT seconds.

When ca_poll is invoked the send buffer is flushed and any outstanding CA background activity is processed.

The ca_pend_event function will *not* return before the specified time-out expires and all unfinished channel access labor has been processed, and unlike `ca_pend_io` returning from the function does *not* indicate anything about the status of pending IO requests.

Both `ca_pend_event` and `ca_poll` 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.

Arguments

`TIMEOUT`
: 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

ca_flush_io()

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

Description

Flush outstanding IO requests to the server. 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.

Returns

ECA_NORMAL - Normal successful completion

ca_signal()

```
#include <cadef.h>
int ca_signal ( long CA_STATUS, const char * CONTEXT_STRING );
```

```
void SEVCHK( CA_STATUS, CONTEXT_STRING );
```

Description

Provide the error message character string associated with the supplied channel access error code and the supplied error context to diagnostics. If the error code indicates an unsuccessful operation a stack dump is printed, if this capability is available on the local operating system, and execution is terminated.

SEVCHK is a macro envelope around ca_signal which only calls ca_signal() if the supplied error code indicates an unsuccessful operation. SEVCHK is the recommended error handler for simple applications which do not wish to write code testing the status returned from each channel access call.

Examples

```
status = ca_context_create (...);
SEVCHK ( status, "Unable to create a CA client context" );
```

If the application only wishes to print the message associated with an error code or test the severity of an error there are also functions provided for this purpose.

Arguments

`CA_STATUS`
: The status (error code) returned from a channel access function.

`CONTEXT_STRING`
: A null terminated character string to supply as error context to diagnostics.

Returns

ECA_NORMAL - Normal successful completion

`ca_add_exception_event()`

```
#include <cadef.h>
typedef void (*pCallback) ( struct exception_handler_args HANDLERARGS );
int ca_add_exception_event ( pCallback  USERFUNC, void *USERARG );
```

Description

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

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 in "struct exception_handler_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.

Arguments

`USERFUNC`
: Address of user callback function to be executed when an exceptions occur. Passing a nil value causes the default exception handler to be reinstalled. The following structure is passed by value to the user's callback function. Currently, the `op` field can be one of `CA_OP_GET, CA_OP_PUT, CA_OP_CREATE_CHANNEL, CA_OP_ADD_EVENT, CA_OP_CLEAR_EVENT, or CA_OP_OTHER.`

```
struct exception_handler_args {
    void            *usr;   /* user argument supplied when installed */
    chanId          chid;   /* channel id (may be nill) */
    long            type;   /* type requested */
    long            count;  /* count requested */
    void            *addr;  /* user's address to write results of CA_OP_GET */
```

```
        long            stat;   /* channel access ECA_XXXX status code */
        long            op;     /* CA_OP_GET, CA_OP_PUT, ..., CA_OP_OTHER */
        const char      *ctx;   /* a character string containing context info */
        sonst char      *pFile; /* source file name (may be NULL) */
        unsigned        lineNo; /* source file line number (may be zero) */
    };
```

USERARG
: pointer sized variable retained and passed back to user function above

Example

```
void ca_exception_handler (
        struct exception_handler_args args)
{
        char buf[512];
        char *pName;

        if ( args.chid ) {
                pName = ca_name ( args.chid );
        }
        else{
                pName = "?";
        }
        sprintf ( buf,
                "%s - with request chan=%s op=%d data type=%s count=%d",
                args.ctx, pName, args.op, dbr_type_to_text ( args.type ), args.count );
        ca_signal ( args.stat, buf );

}
ca_add_exception_event ( ca_exception_handler , 0 );
```

Returns

ECA_NORMAL - Normal successful completion

ca_add_fd_registration()

```
#include <cadef.h>int ca_add_fd_registration ( void ( USERFUNC * ) ( void *USERARG, int FD, int OPENED ), void * USERARG )
```

Description

For use with the services provided by a file descriptor manager (IO multiplexor) such as ""fdmgr.c". A file descriptor manager is often needed when two file descriptor IO intensive libraries such as the EPICS channel access client library and the X window system client library must coexist in the same UNIX process. This function allows an application code to be notified whenever the CA client library places a new file descriptor into service and whenever the CA client library removes a file descriptor from service. Specifying USERFUNC=NULL disables file descriptor registration (this is the default).

Arguments

USERFUNC

Pointer to a user supplied C function returning null with the above arguments.

USERARG

User supplied pointer sized variable passed to the above function.

FD

A file descriptor.

OPENED

Boolean argument is true if the file descriptor was opened and false if the file descriptor was closed.

Example

```
int s;

static struct myStruct aStruct;

void fdReg ( struct myStruct *pStruct, int fd, int opened )

{

    if ( opened ) printf ( "fd %d was opened\n", fd );

    else printf ( "fd %d was closed\n", fd );

}

s = ca_add_fd_registration ( fdReg, & aStruct );

SEVCHK ( s, NULL );
```

Comments

When using this function it is advisable to call it only once prior to calling any other CA function, or once just after creating the CA context (if you create the context explicitly). Use of this interface can improve latency slightly in applications that use non preemptive callback mode at the expense of some additional runtime overhead when compared to the alternative which is just polling ca_pend_event periodically. It would probably not be appropriate to use this function with preemptive callback mode. Starting with R3.14 this function is implemented in a special backward compatibility mode. if `ca_add_fd_registration` is called, a single pseudo UDP fd is created which CA pokes whenever something significant happens. Xt and others can watch this fd so that backwards compatibility is preserved, and so that they will not need to use preemptive callback mode but they will nevertheless get the lowest latency response to the arrival of CA messages.

Returns

"ECA_NORMAL - Normal successful completion

`ca_replace_printf_handler ()`

```
#include <cadef.h>
typedef int caPrintfFunc ( const char *pFromat, va_list args );
int ca_replace_printf_handler ( caPrintfFunc *PFUNC );
```

Description

Replace the default handler for formatted diagnostic message output. The default handler uses fprintf to send messages to 'stderr'.

Arguments

`PFUNC`
: The address of a user supplied call back handler to be invoked when CA prints diagnostic messages. Installing a nil pointer will cause the default call back handler to be reinstalled.

Examples

```
int my_printf ( char *pformat, va_list args ) {
        int status;
        status = vfprintf( stderr, pformat, args);
        return status;
}
status = ca_replace_printf_handler ( my_printf );
SEVCHK ( status, "failed to install my printf handler" );
```

Returns

ECA_NORMAL - Normal successful completion

`ca_replace_access_rights_event()`

```
#include <cadef.h>
typedef void ( *pCallBack )( struct access_rights_handler_args );
int ca_replace_access_rights_event ( chid CHAN, pCallBack PFUNC );
```

Description

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 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.

Arguments

`CHAN`
: The channel identifier.

`PFUNC`
: Address of user supplied call back function. A nil pointer uninstalls the current handler. The following arguments are passed *by value* to the supplied callback handler.

```
typedef struct ca_access_rights {
    unsigned    read_access:1;
    unsigned    write_access:1;
} caar;

/* arguments passed to user access rights handlers */
struct  access_rights_handler_args {
    chanId  chid;   /* channel id */
    caar    ar;     /* new access rights state */
};
```

Returns

ECA_NORMAL - Normal successful completion

See Also

ca_modify_user_name()

ca_modify_host_name()

`ca_field_type()`

```
#include <cadef.h>
chtype ca_field_type ( CHID );
```

Description

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

Arguments

`CHID`
: channel identifier

Returns

`TYPE`
: The data type code will be a member of the set of DBF_XXXX in db_access.h. The constant

TYPENOTCONN is returned if the channel is disconnected.

ca_element_count()

```
#include <cadef.h>
unsigned ca_element_count ( CHID );
```

Description

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

Arguments

CHID
channel identifier

Returns

COUNT
The maximum array element count in the server. An element count of zero is returned if the channel is disconnected.

ca_name()

```
#include <cadef.h>
char * ca_name ( CHID );
```

Description

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

Arguments

CHID
channel identifier

Returns

PNAME
The channel name. The string returned is valid as long as the channel specified exists.

ca_set_puser()

```
#include <cadef.h>
void ca_set_puser ( chid CHID, void *PUSER );
```

Description

Set a user private void pointer variable retained with each channel for use at the users discretion.

Arguments

CHID
channel identifier

PUSER
user private void pointer

ca_puser()

```
#include <cadef.h>
void * ca_puser ( CHID );
```

Description

Return a user private void pointer variable retained with each channel for use at the users discretion.

Arguments

`CHID`
channel identifier

Returns

`PUSER`
user private pointer

`ca_state()`

```
#include <cadef.h>
enum channel_state {
        cs_never_conn, /* valid chid, server not found or unavailable */
        cs_prev_conn,  /* valid chid, previously connected to server */
        cs_conn,       /* valid chid, connected to server */
        cs_closed };   /* channel deleted by user */
enum channel_state ca_state ( CHID );
```

Description

Returns an enumerated type indicating the current state of the specified IO channel.

Arguments

`CHID`
channel identifier

Returns

`STATE`
the connection state

`ca_message()`

```
#include <cadef.h>
const char * ca_message ( STATUS );
```

Description

return a message character string corresponding to a user specified CA status code.

Arguments

`STATUS`
a CA status code

Returns

STRING
the corresponding error message string

`ca_host_name()`

```
#include <cadef.h>
char * ca_host_name ( CHID );
```

Description

Return a character string which contains the name of the host to which a channel is currently connected.

Arguments

`CHID`
: the channel identifier

Returns

`STRING`
: The process variable server's host name. If the channel is disconnected the string "<disconnected>" is returned.

`ca_read_access()`

```
#include <cadef.h>
int ca_read_access ( CHID );
```

Description

Returns boolean true if the client currently has read access to the specified channel and boolean false otherwise.

Arguments

`CHID`
: the channel identifier

Returns

`STRING`
: boolean true if the client currently has read access to the specified channel and boolean false otherwise

`ca_write_access()`

```
#include <cadef.h>
int ca_write_access ( CHID );
```

Description

Returns boolean true if the client currently has write access to the specified channel and boolean false otherwise.

Arguments

`CHID`
: the channel identifier

Returns

`STRING`
: boolean true if the client currently has write access to the specified channel and boolean false otherwise

`dbr_size[]`

```
#include <db_access.h>
extern unsigned dbr_size[/*TYPE*/];
```

Description

An array that returns the size in bytes for a DBR_XXXX type.

Arguments

TYPE
The data type code. A member of the set of DBF_XXXX in db_access.h.

Returns

SIZE
the size in bytes of the specified type

dbr_size_n()

```
#include <db_access.h>
unsigned dbr_size_n  (  TYPE, COUNT  );
```

Description

Returns the size in bytes for a DBR_XXXX type with COUNT elements. If the DBR type is a structure then the value field is the last field in the structure. If COUNT is greater than one then COUNT-1 elements are appended to the end of the structure so that they can be addressed as an array through a pointer to the value field.

Arguments

TYPE
The data type

COUNT
The element count

Returns

SIZE
the size in bytes of the specified type with the specified number of elements

dbr_value_size[]

```
#include <db_access.h>
extern unsigned dbr_value_size[/* TYPE */];
```

Description

The array dbr_value_size[TYPE] returns the size in bytes for the value stored in a DBR_XXXX type. If the type is a structure the size of the value field is returned otherwise the size of the type is returned.

Arguments

TYPE
The data type code. A member of the set of DBF_XXXX in db_access.h.

Returns

SIZE
the size in bytes of the value field if the type is a structure and otherwise the size in bytes of the type

dbr_type_to_text()

```
#include <db_access.h>
const char * dbr_type_text ( chtype TYPE );
```

Description

Returns a constant null terminated string corresponding to the specified dbr type.

Arguments

`TYPE`
The data type code. A member of the set of DBR_XXXX in db_access.h.

Returns

`STRING`
The const string corresponding to the DBR_XXX type.

ca_test_event()

```
#include <cadef.h>
```

Description

```
void ca_test_event ( struct event_handler_args );
```

A built-in subscription update call back handler for debugging purposes that prints diagnostics to standard out.

Examples

```
void ca_test_event ();
status = ca_add_event ( type, chid, ca_test_event, NULL, NULL );
SEVCHK ( status, .... );
```

See Also

ca_add_event()

ca_sg_create()

```
#include <cadef.h>
int ca_sg_create ( CA_SYNC_GID *PGID );
```

Description

Create a synchronous group and return an identifier for it.

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 ca_sg_get() and ca_sg_put() respectively. The routines ca_sg_block() and ca_sg_test() can be used to block for and test for completion respectively. The routine ca_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.

Arguments

`PGID`
Pointer to a user supplied CA_SYNC_GID.

Examples

```
CA_SYNC_GID gid;
status = ca_sg_create ( &gid );
SEVCHK ( status, Sync group create failed );
```

Returns

ECA_NORMAL - Normal successful completion

ECA_ALLOCMEM - Failed, unable to allocate memory

See Also

ca_sg_delete()

ca_sg_block()

ca_sg_test()

ca_sg_reset()

ca_sg_put()

ca_sg_get()

ca_sg_delete()

```
#include <cadef.h>
int ca_sg_delete ( CA_SYNC_GID GID );
```

Description

Deletes a synchronous group.

Arguments

GID
: Identifier of the synchronous group to be deleted.

Examples

```
CA_SYNC_GID gid;
status = ca_sg_delete ( gid );
SEVCHK ( status, Sync group delete failed );
```

Returns

ECA_NORMAL - Normal successful completion

ECA_BADSYNCGRP - Invalid synchronous group

See Also

ca_sg_create()

ca_sg_block()

```
#include <cadef.h>
int ca_sg_block ( CA_SYNC_GID GID, double timeout );
```

Description

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 ca_sg_array_get() and calls to ca_sg_array_put(). If ECA_TIMEOUT is returned then failure must be assumed for all outstanding queries. Operations can be reissued followed by another ca_sg_block(). This routine will only block on outstanding queries issued after the last call to ca_sg_block(), ca_sg_reset(), or ca_sg_create() whichever occurs later in time. If no queries are outstanding then ca_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 ca_sg_block(). This routine will process pending channel access background activity while it is waiting.

Arguments

GID
Identifier of the synchronous group.

Examples

```
CA_SYNC_GID gid;
status = ca_sg_block(gid);
SEVCHK(status, Sync group block failed);
```

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

See Also

ca_sg_test()

ca_sg_reset()

ca_sg_test()

```
#include <cadef.h>
int ca_sg_test  ( CA_SYNC_GID GID )
```

Description

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

Arguments

GID
Identifier of the synchronous group.

Description

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

Examples

```
CA_SYNC_GID gid;
status = ca_sg_test ( gid );
```

Returns

ECA_IODONE - IO operations completed

ECA_IOINPROGRESS - Some IO operations still in progress

ca_sg_reset()

```
#include <cadef.h>
int ca_sg_reset ( CA_SYNC_GID GID )
```

Description

Reset the number of outstanding requests within the specified synchronous group to zero so that ca_sg_test() will return ECA_IODONE and ca_sg_block() will not block unless additional subsequent requests are made.

Arguments

`GID`
: Identifier of the synchronous group.

Examples

```
CA_SYNC_GID gid;
status = ca_sg_reset(gid);
```

Returns

ECA_NORMAL - Normal successful completion

ECA_BADSYNCGRP - Invalid synchronous group

`ca_sg_put()`

```
#include <cadef.h>
int ca_sg_array_put ( CA_SYNC_GID GID, chtype TYPE,
        unsigned long COUNT, chid CHID, void *PVALUE );
```

Write a value, or array of values, to a channel and increment the outstanding request count of a synchronous group. The ca_sg_array_put functionality is implemented using ca_array_put_callback.

All remote operation requests such as the above are accumulated (buffered) and not forwarded to the server until one of ca_flush_io(), ca_pend_io(), ca_pend_event(), or ca_sg_pend() 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.

Arguments

`GID`
: synchronous group identifier

`TYPE`
: The type of supplied value. Conversion will occur if it does not match the native type. Specify one from the set of DBR_XXXX in db_access.h.

`COUNT`
: element count to be written to the specified channel - must match the array pointed to by PVALUE

`CHID`
: channel identifier

`PVALUE`
: A pointer to an application supplied buffer containing the value or array of values returned

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

See Also

ca_flush_io()

ca_sg_get()

```
#include <cadef.h>
int ca_sg_array_get ( CA_SYNC_GID GID,
        chtype TYPE, unsigned long COUNT,
        chid CHID, void *PVALUE );
```

Description

Read a value from a channel and increment the outstanding request count of a synchronous group. The ca_sg_array_get functionality is implemented using ca_array_get_callback.

The values written into your program's variables by ca_sg_get should not be referenced by your program until ECA_NORMAL has been received from ca_sg_block , or until ca_sg_test returns ECA_IODONE.

All remote operation requests such as the above are accumulated (buffered) and not forwarded to the server until one of ca_flush_io, ca_pend_io, ca_pend_event, or ca_sg_pend 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.

Arguments

`GID`
: Identifier of the synchronous group.

`TYPE`
: External type of returned value. Conversion will occur if this does not match native type. Specify one from the set of DBR_XXXX in db_access.h

`COUNT`
: Element count to be read from the specified channel. It must match the array pointed to by PVALUE.

`CHID`
: channel identifier

`PVALUE`
: Pointer to application supplied buffer that is to contain the value or array of values to be returned

Returns

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

See Also

ca_pend_io()

ca_flush_io()

ca_get_callback()

ca_client_status()

```
int ca_client_status ( unsigned level );
int ca_context_status ( struct ca_client_context *,
       unsigned level );
```

Description

Prints information about the client context including, at higher interest levels, status for each channel. Lacking a CA context pointer, ca_client_status() prints information about the calling threads CA context.

Arguments

`CONTEXT`
: A pointer to the CA context to join with.

`LEVEL`
: The interest level. Increasing level produces increasing detail.

ca_current_context()

```
struct ca_client_context * ca_current_context ();
```

Description

Returns a pointer to the current thread's CA context. If none then nil is returned.

See Also

ca_attach_context()

ca_detach_context()

ca_context_create()

ca_context_destroy()

ca_attach_context()

```
int ca_attach_context (struct ca_client_context *CONTEXT);
```

Description

The calling thread becomes a member of the specified CA context. If `ca_disable_preemptive_callback` is specified when ca_context_create() is called (or if ca_task_initialize() is called) 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.

Arguments

`CONTEXT`
: A pointer to the CA context to join with.

Returns

ECA_ISATTACHED - already attached to a CA context

ECA_NOTTHREADED - the specified context is non-preemptive and therefore does not allow other threads to join

ECA_ISATTACHED - the current thread is already attached to a CA context

See Also

ca_current_context()

ca_detach_context()

ca_context_create()

ca_context_destroy()

ca_detach_context()

```
void ca_detach_context();
```

Description

Detach from any CA context currently attached to the calling thread. This does *not* cleanup or shutdown any currently attached CA context (for that use ca_context_destroy).

See Also

ca_current_context()

ca_attach_context()

ca_context_create()

ca_context_destroy()

ca_dump_dbr()

```
void ca_dump_dbr ( chtype TYPE, unsigned COUNT, const void * PDBR );
```

Description

Dumps the specified dbr data type to standard out.

Arguments

`TYPE`
: The data type (from the DBR_XXX set described in db_access.h).

`COUNT`
: The array element count

`PDBR`
: A pointer to data of the specified count and number.

Return Codes

ECA_NORMAL
: Normal successful completion

ECA_ALLOCMEM
: Unable to allocate additional dynamic memory

ECA_TOLARGE
: The requested data transfer is greater than available memory or EPICS_CA_MAX_ARRAY_BYTES

ECA_BADTYPE
: The data type specified is invalid

ECA_BADSTR

Invalid string
ECA_BADCHID
Invalid channel identifier
ECA_BADCOUNT
Invalid element count requested
ECA_PUTFAIL
Channel write request failed
ECA_GETFAIL
Channel read request failed
ECA_ADDFAIL
unable to install subscription request
ECA_TIMEOUT
User specified timeout on IO operation expired
ECA_EVDISALLOW
function called was inappropriate for use within a callback function
ECA_IODONE
IO operations have completed
ECA_IOINPROGRESS
IO operations are in progress
ECA_BADSYNCGRP
Invalid synchronous group identifier
ECA_NORDACCESS
Read access denied
ECA_NOWTACCESS
Write access denied
ECA_DISCONN
Virtual circuit disconnect"
ECA_DBLCHNL
Identical process variable name on multiple servers
ECA_EVDISALLOW
Request inappropriate within subscription (monitor) update callback
ECA_BADMONID
Bad event subscription (monitor) identifier
ECA_BADMASK
Invalid event selection mask
ECA_PUTCBINPROG
Put callback timed out
ECA_PUTCBINPROG
Put callback timed out
ECA_ANACHRONISM
Requested feature is no longer supported
ECA_NOSEARCHADDR
Empty PV search address list
ECA_NOCONVERT
No reasonable data conversion between client and server types
ECA_BADFUNCPTR
Invalid function pointer
ECA_ISATTACHED
Thread is already attached to a client context
ECA_UNAVAILINSERV
Not supported by attached service
ECA_CHANDESTROY
User destroyed channel
ECA_BADPRIORITY
Invalid channel priority
ECA_NOTTHREADED
Preemptive callback not enabled - additional threads may not join context
ECA_16KARRAYCLIENT
Client's protocol revision does not support transfers exceeding 16k bytes

$Id: CAref.html,v 1.58.2.51 2009/07/30 23:09:54 jhill Exp $.