



# Introduction to Channel Access Client Library

**Kazuro Furukawa**

<kazuro.furukawa@kek.jp>

for EPICS2010 at IHEP/Beijing

March 3, 2010

Based on presentations by

Kenneth Evans, Jr., 2004

Kazuro Furukawa, 2006

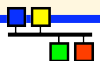
Kay Kasemir, 2007





# Outline

- ◆ **Channel Access Concepts**
- ◆ **Channel Access API**
- ◆ **Simple CA Client**
- ◆ **Simple CA Client with Callbacks**
- ◆ **(EPICS Build System)**



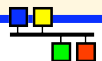
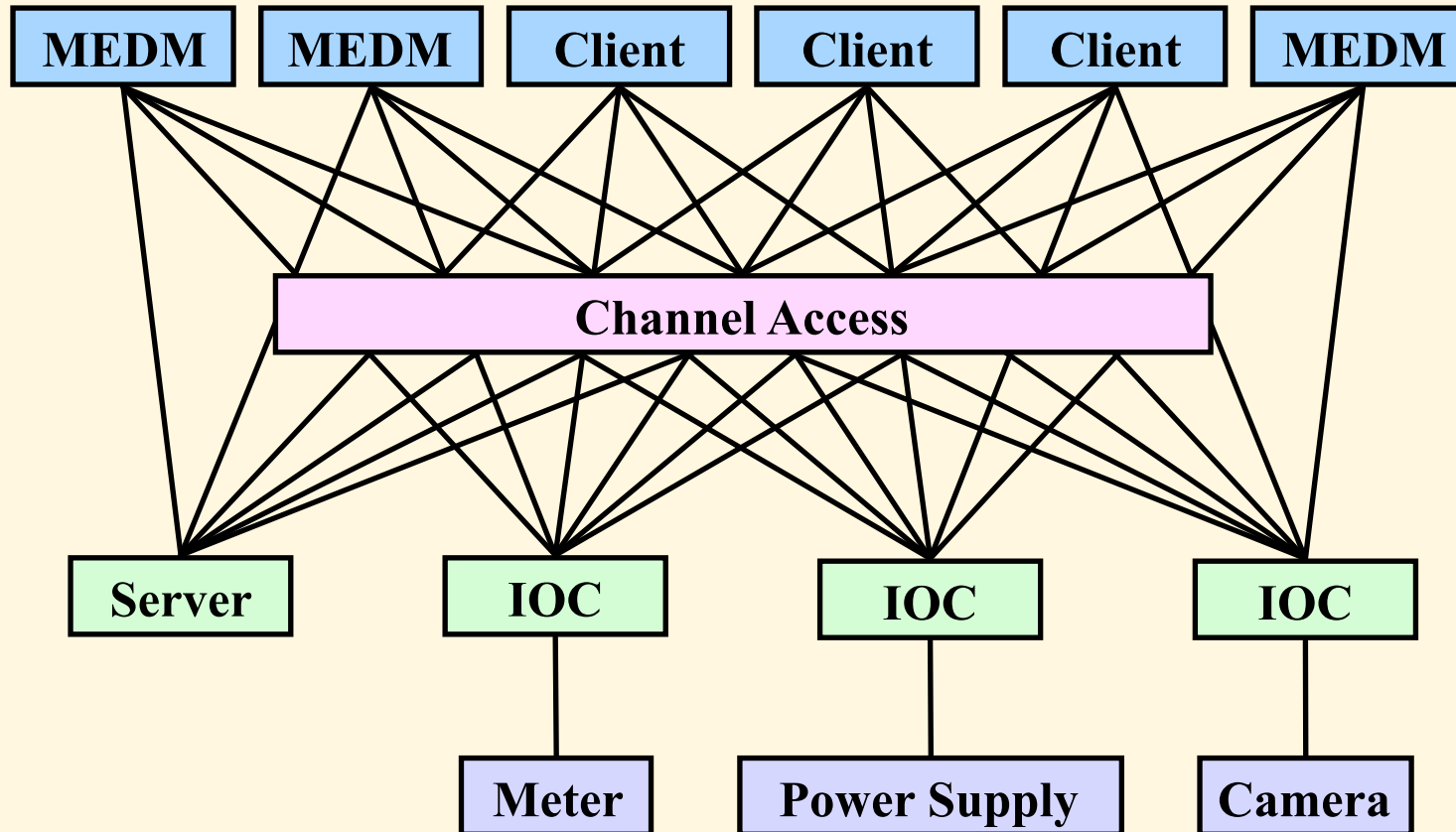


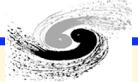
# Channel Access Reference Manual

- ◆ The place to go for more information
- ◆ Found in the EPICS web pages
  - ❖ <http://www.aps.anl.gov/epics/index.php>
  - ❖ Look under Documents
  - ❖ Also under Base, then a specific version of Base

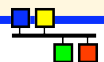
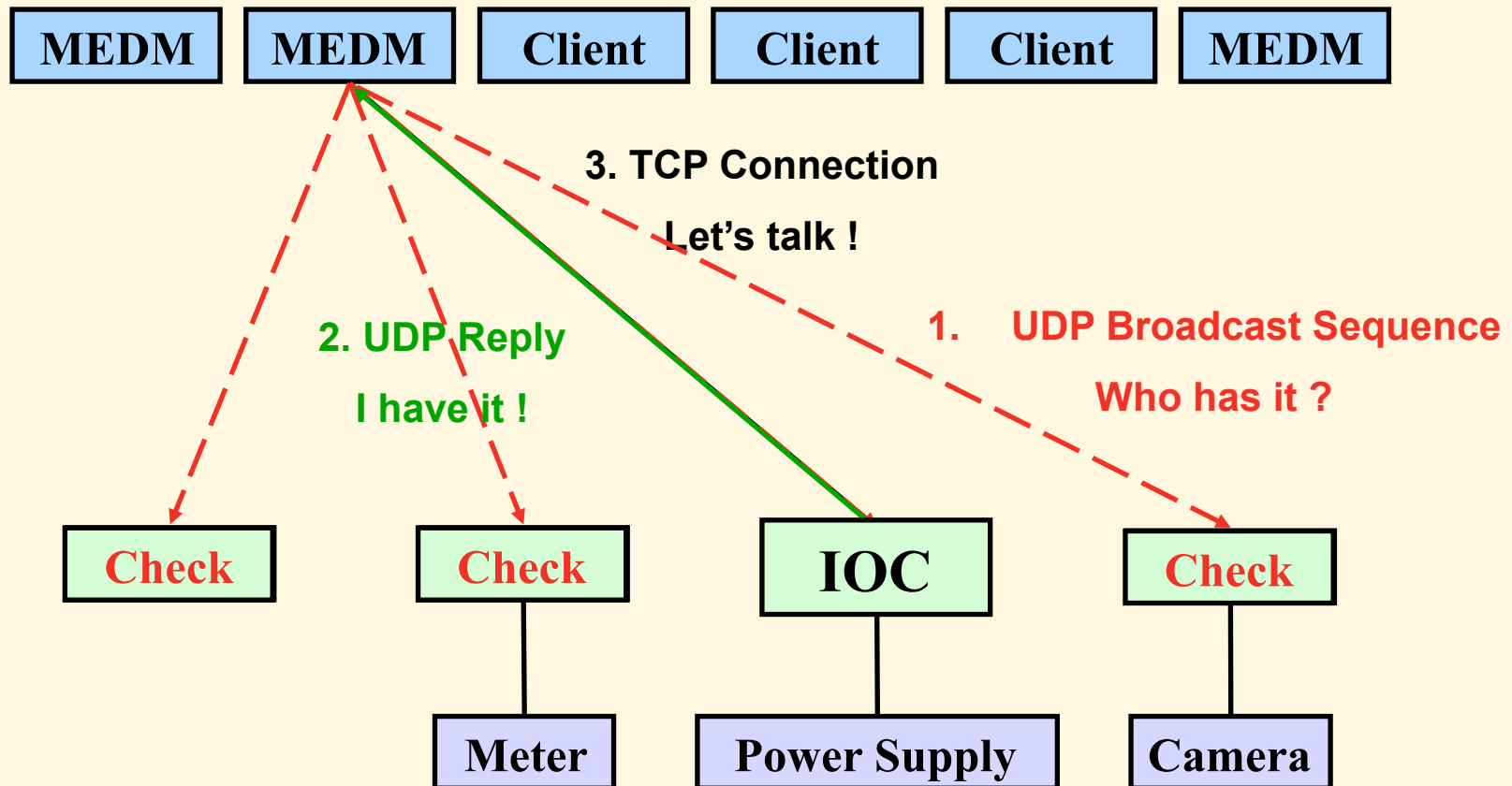


# EPICS Overview



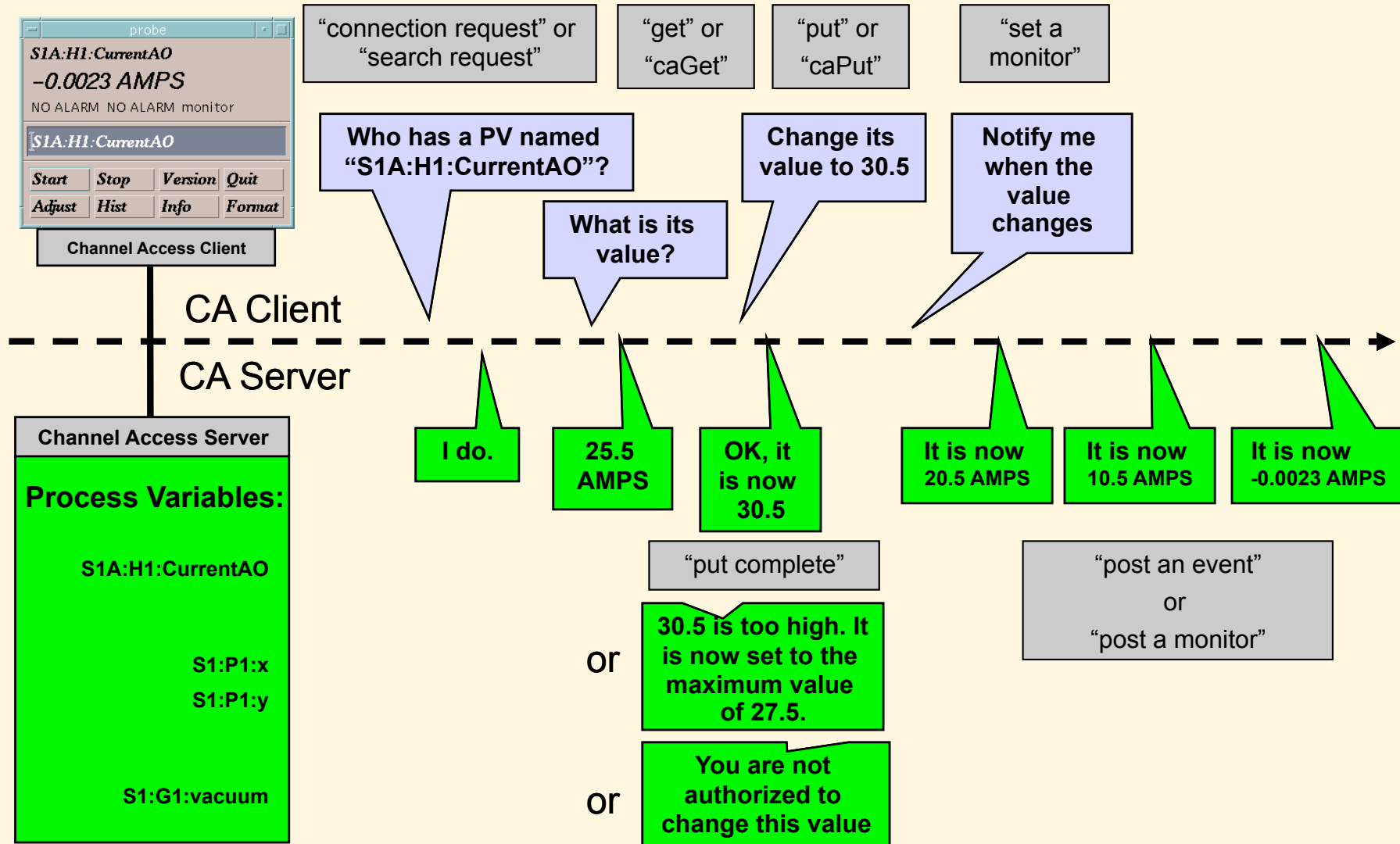


# Search and Connect Procedure





# Channel Access in One Slide





# Search Request

- ◆ **A search request consists of a sequence of UDP packets**
  - ❖ Only goes to EPICS\_CA\_ADDR\_LIST
  - ❖ Starts with a small interval (30 ms), that doubles each time
  - ❖ Until it gets larger than 5 s, then it stays at 5 s
  - ❖ Stops after 100 packets or when it gets a response
  - ❖ Never tries again until it sees a beacon anomaly or creates a new PV
  - ❖ Total time is about 8 minutes to do all 100



- ◆ **Servers have to do an Exist Test for each packet**
- ◆ **Usually connects on the first packet or the first few**
- ◆ **Non-existent PVs cause a lot of traffic**
  - ❖ Try to eliminate them



# Beacons

- ◆ A Beacon is a UDP broadcast packet sent by a Server
- ◆ When it is healthy, each Server broadcasts a UDP beacon at regular intervals (like a heartbeat)
  - ❖ EPICS\_CA\_BEACON\_PERIOD, 15 s by default
- ◆ When it is coming up, each Server broadcasts a startup sequence of UDP beacons
  - ❖ Starts with a small interval (25 ms, 75 ms for VxWorks)
  - ❖ Interval doubles each time
  - ❖ Until it gets larger than 15 s, then it stays at 15 s
    - ✧ Takes about 10 beacons and 40 s to get to steady state
- ◆ Clients monitor the beacons
  - ❖ Determine connection status, whether to reissue searches





# Virtual Circuit Disconnect

## ◆ 3.13 and early 3.14

- ❖ Hang-up message or no response from server for 30 sec.
- ❖ If not a hang-up, then client sends “Are you there” query
- ❖ If no response for 5 sec, TCP connection is closed
- ❖ MEDM screens go white
- ❖ Clients reissue search requests

## ◆ 3.14 5 and later

- ❖ Hang-up message from server
- ❖ TCP connection is closed
- ❖ MEDM screens go white
- ❖ Clients reissue search requests



# Virtual Circuit Unresponsive

## ◆ 3.14.5 and later

- ❖ No response from server for 30 sec.
- ❖ Client then sends “Are you there” query
- ❖ If no response for 5 sec, TCP connection is **not** closed
  - ✧ For several hours, at least
- ❖ MEDM screens go white
- ❖ Clients **do not** reissue search requests
  - ✧ Helps with network storms
  
- ❖ Clients that do not call `ca_poll` frequently get a virtual circuit disconnect even though the server may be OK
  - ✧ Clients written for 3.13 but using 3.14 may have a problem
  - ✧ May be changed in future versions



# Important Environment Variables

## ◆ EPICS\_CA\_ADDR\_LIST

- ❖ Determines where to search
- ❖ Is a list (separated by spaces)
  - ✧ “123.45.1.255 123.45.2.14 123.45.2.108”
- ❖ Default is broadcast addresses of all interfaces on the host
  - ✧ Works when servers are on same subnet as Clients
- ❖ Broadcast address
  - ✧ Goes to all servers on a subnet
  - ✧ Example: 123.45.1.255
  - ✧ Use `ifconfig -a` on UNIX to find it (or ask an administrator)

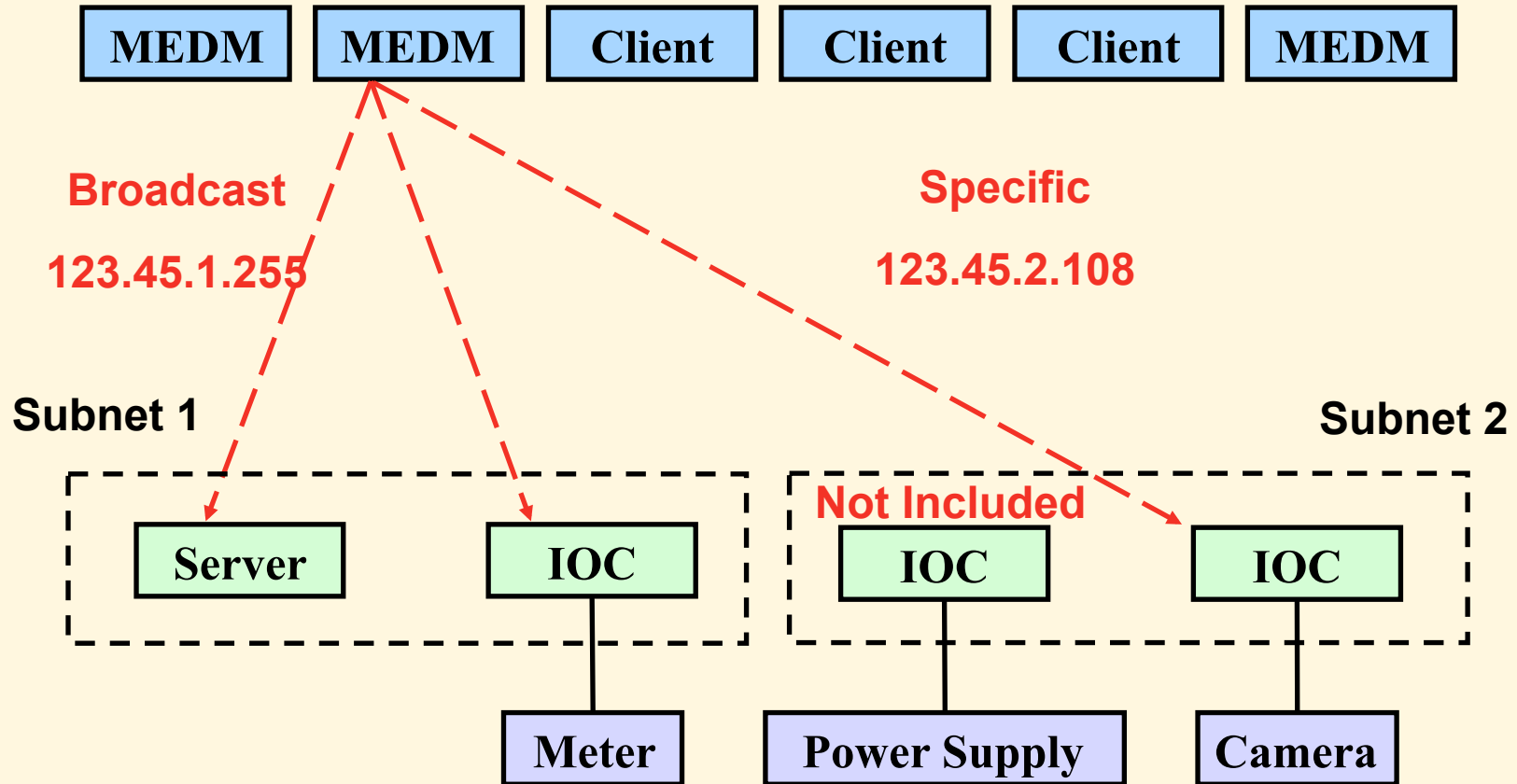
## ◆ EPICS\_CA\_AUTO\_ADDR\_LIST

- ❖ YES: Include default addresses above in searches
- ❖ NO: Do not search on default addresses
- ❖ If you set EPICS\_CA\_ADDR\_LIST, usually set this to NO





# EPICS\_CA\_ADDR\_LIST





# Other Environment Variables

## ◆ CA Client

EPICS\_CA\_ADDR\_LIST  
EPICS\_CA\_AUTO\_ADDR\_LIST  
EPICS\_CA\_CONN\_TMO  
EPICS\_CA\_BEACON\_PERIOD  
EPICS\_CA\_REPEATER\_PORT  
EPICS\_CA\_SERVER\_PORT  
EPICS\_CA\_MAX\_ARRAY\_BYTES  
EPICS\_TS\_MIN\_WEST

## ◆ CA Server

EPICS\_CAS\_SERVER\_PORT  
EPICS\_CAS\_AUTO\_BEACON\_ADDR\_LIST  
EPICS\_CAS\_BEACON\_ADDR\_LIST  
EPICS\_CAS\_BEACON\_PERIOD  
EPICS\_CAS\_BEACON\_PORT  
EPICS\_CAS\_INTF\_ADDR\_LIST  
EPICS\_CAS\_IGNORE\_ADDR\_LIST

◆ See the Channel Access Reference Manual for more information



## 3.13 and 3.14 Similarities

- ◆ Much effort has done into making clients written for 3.13 work with 3.14 with no changes to the coding
- ◆ Even large programs like MEDM have had to make only a few minor changes
- ◆ This means existing programs typically do not need to be rewritten
  - ❖ **This is good!**
- ◆ In contrast, Channel Access Servers require many changes in converting to 3.14



## 3.13 and 3.14 Differences

- ◆ **3.14 is threaded**
  - ❖ Your program does not have to be threaded
- ◆ **3.14 has different names for some functions**
  - ❖ `ca_context_create` for `ca_task_initialize`
  - ❖ `ca_context_destroy` for `ca_task_exit`
  - ❖ `ca_create_channel` for `ca_search_and_connect`
  - ❖ `ca_create_subscription` for `ca_add_event`
  - ❖ `ca_clear_subscription` for `ca_clear_event`
  - ❖ The new functions may have more capabilities, usually related to threading
  - ❖ We will use the new names
- ◆ **3.14 has a different mechanism for lost connections**
  - ❖ Virtual circuit unresponsive (Not available in 3.13)
  - ❖ Virtual circuit disconnected



# Channel Access

- ◆ The main CA client interface is the "C" library that comes with EPICS base
  - ❖ Internally uses C++, but API is pure C.
- ◆ Almost all other CA client interfaces use that C library
  - ❖ Exception: New pure Java JAC







## Basic Procedure for a Channel Access Client

- ◆ **Initialize Channel Access**
  - ❖ `ca_task_initialize` or `ca_context_create`
- ◆ **Search**
  - ❖ `ca_search_and_connect` or `ca_create_channel`
- ◆ **Do get or put**
  - ❖ `ca_get` or `ca_put`
- ◆ **Monitor**
  - ❖ `ca_add_event` or `ca_create_subscription`
- ◆ **Give Channel Access a chance to work**
  - ❖ `ca_poll`, `ca_pend_io`, `ca_pend_event`
- ◆ **Clear a channel**
  - ❖ `ca_clear_channel`
- ◆ **Close Channel Access**
  - ❖ `ca_task_exit` or `ca_context_destroy`



# caodef.h

- ◆ **All C or C++ programs must include caodef.h**

  - ❖ `#include <caodef.h>`

- ◆ **You can look at this file to get more insight into Channel Access**

- ◆ **This presentation will use C examples**

  - ❖ **We will try to emphasize concepts, not the language**

  - ❖ **Even if you do not use C, it is important to understand what is going on behind what you do use**





# ca\_context\_create

```
enum ca_preemptive_callback_select {
    ca_disable_preemptive_callback,
    ca_enable_preemptive_callback };
int ca_context_create (
    enum ca_preemptive_callback_select SELECT );
```

- ◆ Should be called once prior to any other calls
- ◆ Sets up Channel Access
- ◆ Use **SELECT**=ca\_disable\_preemptive\_callback
  - ❖ Unless you intend to do threads
- ◆ Can also use ca\_task\_initialize() for 3.13 compatibility





# ca\_context\_destroy

```
void ca_context_destroy ();
```

- ◆ Should be called before exiting your program
- ◆ Shuts down Channel Access
- ◆ Can also use `ca_task_exit()` for 3.13 compatibility





# ca\_create\_channel

```
typedef void caCh (struct connection_handler_args ARGS) ;  
int ca_create_channel (  
    const char *PVNAME ,  
    caCh *CALLBACK ,  
    void *PUSER ,  
    capri PRIORITY ,  
    chid *PCHID ) ;
```

- ◆ Sets up a channel and starts the search process
- ◆ **PVNAME** is the name of the process variable
- ◆ **CALLBACK** is the name of your connection callback (or NULL)
  - ❖ The callback will be called whenever the connection state changes, including when first connected
  - ❖ Information about the channel is contained in **ARGS**
  - ❖ Use NULL if you don't need a callback





# ca\_create\_channel, cont'd

```
typedef void caCh (struct connection_handler_args ARGS) ;  
int ca_create_channel (  
    const char *PVNAME ,  
    caCh *CALLBACK ,  
    void *PUSER ,  
    capri PRIORITY ,  
    chid *PCHID ) ;
```

## ◆ **PUSER** is a way to pass additional information

- ❖ Whatever you have stored at this address
- ❖ It is stored in the `chid`
- ❖ In C++ it is often the **this** pointer for a class
- ❖ Use NULL if you don't need it

## ◆ Use **PRIORITY=CA\_PRIORITY\_DEFAULT**





# ca\_create\_channel, cont'd

```
typedef void caCh (struct connection_handler_args ARGS);  
int ca_create_channel (  
    const char *PVNAME,  
    caCh *CALLBACK,  
    void *PUSER,  
    capri PRIORITY,  
    chid *PCHID );
```

◆ A `chid` is a pointer to (address of) an opaque `struct` used by Channel Access to store much of the channel information

❖ `chanId` is the same as `chid` (`typedef chid chanId;`)

◆ `PCHID` is the address of the `chid` pointer (Use `&CHID`)

❖ You need to allocate space for the `chid` before making the call

❖ Channel Access will allocate space for the `struct` and return the address



# ca\_create\_channel, cont'd

```
typedef void caCh (struct connection_handler_args ARGS);  
int ca_create_channel (  
    const char *PVNAME,  
    caCh *CALLBACK,  
    void *PUSER,  
    capri PRIORITY,  
    chid *PCHID );
```

## ◆ Use macros to access the information in the *chid*

- ❖ *ca\_name(CHID)* gives the process variable name
- ❖ *ca\_state(CHID)* gives the connection state
- ❖ *ca\_puser(CHID)* gives the **PUSER** you specified
- ❖ *Etc.*

## ◆ The **ARGS** struct in the connection callback includes the *chid*

## ◆ Can also use *ca\_search\_and\_connect()* for 3.13 compatibility





## ca\_clear\_channel

```
int ca_clear_channel (chid CHID);
```

- ◆ Shuts down a channel and reclaims resources
- ◆ Should be called before exiting the program
- ◆ **CHID** is the same `chid` used in `ca_create_channel`





# ca\_array\_get

```
int ca_array_get (  
    ctype TYPE,  
    unsigned long COUNT,  
    chid CHID,  
    void *PVALUE );
```

- ◆ Requests a scalar or array value from a process variable
- ◆ Typically followed by `ca_pend_io`
- ◆ **TYPE** is the external type of your variable
  - ❖ Use one of the `DBR_XXX` types in `db_access.h`
  - ❖ E.g. `DBR_DOUBLE` or `DBR_STRING`
- ◆ **COUNT** is the number of array elements to read
- ◆ **CHID** is the channel identifier from `ca_create_channel`
- ◆ **PVALUE** is where you want the value(s) to go
  - ❖ There must be enough space to hold the values

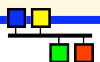




# ca\_array\_get\_callback

```
typedef void ( *pCallback ) (struct event_handler_args ARGS);  
int ca_array_get_callback (  
    ctype TYPE,  
    unsigned long COUNT,  
    chid CHID,  
    pCallback USERFUNC,  
    void *USERARG );
```

- ◆ Requests a scalar or array value from a process variable, using a callback
- ◆ **TYPE** is the external type of your variable
  - ❖ Use one of the `DBR_XXX` types in `db_access.h`
  - ❖ E.g. `DBR_DOUBLE` or `DBR_STRING`
- ◆ **COUNT** is the number of array elements to read





## ca\_array\_get\_callback, cont'd

```
typedef void ( *pCallback ) (struct event_handler_args ARGS) ;  
int ca_array_get_callback (   
    ctype TYPE ,   
    unsigned long COUNT ,   
    chid CHID ,   
    pCallback USERFUNC ,   
    void *USERARG ) ;
```

- ◆ **CHID** is the channel identifier from `ca_create_channel`
- ◆ **USERFUNC** is the name of your callback to be run when the operation completes
- ◆ **USERARG** is a way to pass additional information to the callback
  - ❖ `struct event_handler_args` has a `void *usr` member



# ca\_array\_put

```
int ca_array_put (  
    ctype TYPE,  
    unsigned long COUNT,  
    chid CHID,  
    const void *PVALUE);
```

- ◆ Requests writing a scalar or array value to a process variable
- ◆ Typically followed by `ca_pend_io`
- ◆ **TYPE** is the external type of your supplied variable
  - ❖ Use one of the `DBR_XXX` types in `db_access.h`
  - ❖ E.g. `DBR_DOUBLE` or `DBR_STRING`
- ◆ **COUNT** is the number of array elements to write
- ◆ **CHID** is the channel identifier from `ca_create_channel`
- ◆ **PVALUE** is where the value(s) to be written are found





# ca\_array\_put\_callback

```
typedef void ( *pCallback ) (struct event_handler_args ARGS) ;  
int ca_array_put_callback (   
    ctype TYPE ,   
    unsigned long COUNT ,   
    chid CHID ,   
    const void *PVALUE ,   
    pCallback USERFUNC ,   
    void *USERARG ) ;
```

◆ Requests writing a scalar or array value to a process variable, using a callback

◆ **TYPE** is the external type of your variable

❖ Use one of the `DBR_XXX` types in `db_access.h`

❖ E.g. `DBR_DOUBLE` or `DBR_STRING`





## ca\_array\_put\_callback, cont'd

```
typedef void ( *pCallback ) (struct event_handler_args ARGS );  
int ca_array_put_callback (   
    ctype TYPE,  
    unsigned long COUNT,  
    chid CHID,  
    const void *PVALUE,  
    pCallback USERFUNC,  
    void *USERARG );
```

- ◆ **COUNT** is the number of array elements to write
- ◆ **CHID** is the channel identifier from `ca_create_channel`
- ◆ **PVALUE** is where the value(s) to be written are found





## ca\_array\_put\_callback, cont'd

```
typedef void ( *pCallback ) (struct event_handler_args ARGS );  
int ca_array_put_callback (   
    ctype TYPE,  
    unsigned long COUNT,  
    chid CHID,  
    const void *PVALUE,  
    pCallback USERFUNC,  
    void *USERARG );
```

◆ **USERFUNC** is the name of your callback to be run when the operation completes

◆ **USERARG** is a way to pass additional information to the callback

❖ `struct event_handler_args` has a `void *usr` member





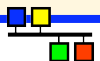


# ca\_create\_subscription

```
typedef void ( *pCallback ) (struct event_handler_args ARGS);  
int ca_create_subscription (  
    ctype TYPE,  
    unsigned long COUNT,  
    chid CHID,  
    unsigned long MASK,  
    pCallback USERFUNC,  
    void *USERARG,  
    evid *PEVID );
```

◆ Specify a callback function to be invoked whenever the process variable undergoes significant state changes

- ❖ Value, Alarm status, Alarm severity
- ❖ This is the way to monitor a process variable





## ca\_create\_subscription, cont'd

```
typedef void ( *pCallback ) (struct event_handler_args ARGS );  
int ca_create_subscription (   
    ctype TYPE,  
    unsigned long COUNT,  
    chid CHID,  
    unsigned long MASK,  
    pCallback USERFUNC,  
    void *USERARG,  
    evid *PEVID );
```

- ◆ **TYPE** is the external type you want returned
  - ❖ Use one of the `DBR_XXX` types in `db_access.h`
  - ❖ E.g. `DBR_DOUBLE` or `DBR_STRING`
- ◆ **COUNT** is the number of array elements to monitor





# ca\_create\_subscription, cont'd

```
typedef void ( *pCallback ) (struct event_handler_args ARGS) ;  
int ca_create_subscription (   
    ctype TYPE,  
    unsigned long COUNT,  
    chid CHID,  
    unsigned long MASK,  
    pCallback USERFUNC,  
    void *USERARG,  
    evid *PEVID ) ;
```

- ◆ **CHID** is the channel identifier from `ca_create_channel`
- ◆ **MASK** has bits set for each of the event trigger types requested
  - ❖ `DBE_VALUE`    Value changes
  - ❖ `DBE_LOG`                    Exceeds archival deadband
  - ❖ `DBE_ALARM`    Alarm state changes





## ca\_create\_subscription, cont'd

```
typedef void ( *pCallback ) (struct event_handler_args ARGS) ;  
int ca_create_subscription (   
    ctype TYPE ,  
    unsigned long COUNT ,  
    chid CHID ,  
    unsigned long MASK ,  
    pCallback USERFUNC ,  
    void *USERARG ,  
    evid *PEVID ) ;
```

◆ **USERFUNC** is the name of your callback to be run when the state change occurs

◆ **USERARG** is a way to pass additional information to the callback

❖ `struct event_handler_args` has a `void *usr` member



## ca\_create\_subscription, cont'd

```
typedef void ( *pCallback ) (struct event_handler_args ARGS) ;  
int ca_create_subscription (   
    ctype TYPE ,   
    unsigned long COUNT ,   
    chid CHID ,   
    unsigned long MASK ,   
    pCallback USERFUNC ,   
    void *USERARG ,   
    evid *PEVID ) ;
```

- ◆ **PEVID** is the address of an `evid` (event id)
  - ❖ You need to allocate space for the `evid` before making the call
  - ❖ Similar to a `chid`
  - ❖ Only used to clear the subscription (Can be NULL if not needed)



# ca\_clear\_subscription

```
int ca_clear_subscription ( evid EVID );
```

- ◆ Used to remove a monitor callback
- ◆ **EVID** is the `evid` from `ca_create_subscription`



# ca\_add\_exception\_event

```
typedef void (*pCallback) (struct exception_handler_args ARGS );  
int ca_add_exception_event (  
    pCallback USERFUNC,  
    void *USERARG );
```

- ◆ Used to replace the default exception handler
- ◆ **USERFUNC** is the name of your callback to be run when an exception occurs
  - ❖ Use **NULL** to remove the callback
- ◆ **USERARG** is a way to pass additional information to the callback
  - ❖ `struct exception_handler_args` has a `void *usr` member





# Request Handling

- ◆ The preceding routines are *requests*
  - ❖ They only queue the operation
  - ❖ They hardly ever fail
    - ✧ The return values are almost always `ECA_NORMAL`
    - ✧ (But they should be checked)
- ◆ These requests are only processed when one of the following is called
  - ❖ `ca_pend_io`                      Blocks until requests are processed
  - ❖ `ca_pend_event`                      Blocks a specified time
  - ❖ `ca_poll`                              Processes current work only
- ◆ If these routines are not called, the requests are not processed and background tasks are also not processed
- ◆ The rule is that one of these should be called every 100 ms
  - ❖ To allow processing of background tasks (beacons, etc.)







# ca\_pend\_io

```
int ca_pend_io (double TIMEOUT);
```

- ◆ Flushes the send buffer
- ◆ Blocks for up to **TIMEOUT** seconds until
  - ❖ Outstanding gets complete
  - ❖ Searches with no callback have connected
- ◆ Returns **ECA\_NORMAL** when gets and searches are complete
- ◆ Returns **ECA\_TIMEOUT** otherwise
  - ❖ Means something went wrong
  - ❖ Get requests can be reissued
  - ❖ Search requests can be reissued after **ca\_clear\_channel**
- ◆ Channel Access background tasks are performed
  - ❖ Unless there were no outstanding I/O requests
- ◆ Use with searches, gets, and puts that don't use callbacks





# ca\_pend\_event

```
int ca_pend_event (double TIMEOUT) ;
```

- ◆ Flushes the send buffer
- ◆ Process background tasks for **TIMEOUT** seconds
  - ✦ Does not return until **TIMEOUT** seconds have elapsed
- ◆ Use this when your application doesn't have to do anything else
  
- ◆ Use `ca_pend_event` instead of `sleep`





# ca\_poll

```
int ca_poll ();
```

- ◆ Flushes the send buffer
- ◆ Process outstanding tasks only
  - ✦ Exits when there are no more outstanding tasks
    - ✧ Otherwise similar to `ca_pend_event`
- ◆ Use this when your application has other things to do
  - ✦ E.g. most GUI programs
- ◆ Be sure it is called at least every 100 ms





# CHID Macros

```
ctype ca_field_type ( CHID );
unsigned ca_element_count ( CHID );
char *ca_name ( CHID );
void *ca_puser ( CHID );
void ca_set_puser ( chid CHID, void *PUSER );
enum channel_state ca_state ( CHID );
    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
char *ca_host_name ( CHID );
int ca_read_access ( CHID );
int ca_write_access ( CHID );
```



# ca\_connection\_handler\_args

```
struct ca_connection_handler_args {  
    chanId chid;          Channel id  
    long op;             CA_OP_CONN_UP or  
                        CA_OP_CONN_DOWN  
};
```

- ◆ Used in connection callback
- ◆ Note chanId is used rather than chid
  - ❖ Some compilers don't like `chid chid;`





# event\_handler\_args

```
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  
} evargs;
```

- ◆ Used in get, put, and monitor callbacks
- ◆ Do not use the value in dbr if status is not ECA\_NORMAL





# Channel Access API Functions

**ca\_add\_exception\_event**  
**ca\_attach\_context**  
**ca\_clear\_channel**  
**ca\_clear\_subscription**  
**ca\_client\_status**  
**ca\_context\_create**  
**ca\_context\_destroy**  
**ca\_context\_status**  
**ca\_create\_channel**  
**ca\_create\_subscription**  
**ca\_current\_context**  
**ca\_dump\_dbr()**  
**ca\_element\_count**  
**ca\_field\_type**  
**ca\_flush\_io**

**ca\_get**  
**ca\_host\_name**  
**ca\_message**  
**ca\_name**  
**ca\_read\_access**  
**ca\_replace\_access\_rights\_event**  
**ca\_replace\_printf\_handler**  
**ca\_pend\_event**  
**ca\_pend\_io**  
**ca\_poll**  
**ca\_puser**  
**ca\_put**  
**ca\_set\_puser**  
**ca\_signal**  
**ca\_sg\_block**  
**ca\_sg\_create**

**ca\_sg\_delete**  
**ca\_sg\_get**  
**ca\_sg\_put**  
**ca\_sg\_reset**  
**ca\_sg\_test**  
**ca\_state**  
**ca\_test\_event**  
**ca\_test\_io**  
**ca\_write\_access**  
**channel\_state**  
**dbr\_size[]**  
**dbr\_size\_n**  
**dbr\_value\_size[]**  
**dbr\_type\_to\_text**  
**SEVCHK**

## Deprecated

**ca\_add\_event**  
**ca\_clear\_event**

**ca\_search**  
**ca\_search\_and\_connect**

**ca\_task\_exit**  
**ca\_task\_initialize**





# Ubuntu Test Network Setup

## ◆ Warning

- ❖ Bellow should be tried without real network
- ❖ Normally, you should follow the institute rules
- ❖ Running “excas” all the time is bad

## ◆ Boot Ubuntu Linux from USB memory

## ◆ Applications -> Accessories -> Terminal

- ❖ `sudo ifconfig eth0 192.168.1.2`
- ❖ `ping 192.168.1.2`
- ❖ `excas -p z: &`
- ❖ `caget z:fred z:freddy z:jane z:janet`

## ◆ After you finish

- ❖ `killall excas`





# makeBaseApp.pl

## ◆ Includes a template for basic CA client in C:

### ❖ Start with this:

```
echo $PATH $EPICS_HOST_ARCH $EPICS_CA_ADDR_LIST  
mkdir cac ; cd cac  
makeBaseApp.pl -t caClient cacApp  
make 2>&1 | tee make.log
```

### ❖ Result:

```
bin/linux-x86/caExample <some PV>  
bin/linux-x86/caMonitor <file with PV list>
```

- ❖ cd bin/linux-x86
- ❖ ./caExample z:fred
- ❖ echo z:fred > alist
- ❖ echo z:freddy >> alist
- ❖ ./caMonitor alist

### ❖ Then read the sources, compare with the reference manual, and edit/extend to suit your needs.



# makeBaseApp's caExample.c

## ◆ Minimal CA client program.

- ❖ Fixed timeout, waits until data arrives.

- ❖ Requests everything as 'DBR\_DOUBLE'.

  - ✧ ... which results in values of C-type 'double'.

  - ✧ See db\_access.h header file for all the DBR\_... constants and the resulting C types or structures.

  - ✧ In addition to the basic DBR\_<type> requests, it is possible to request packaged attributes like DBR\_CTRL\_<type> to get { value, units, limits, ...} in one request.



# makeBaseApp's caMonitor.c

## ◆ Better CA client program.

- ❖ Registers callbacks to get notified when connected or disconnected
- ❖ Subscribes to value updates instead of waiting.
- ❖ ... but still uses the same data type (DBR\_STRING) for everything.





# Ideal CA client?

- ◆ **Use callbacks for everything**
  - ❖ no idle 'wait', no fixed time outs.
- ◆ **Upon connection, check the channel's *native* type (int, double, string, ...)**
  - ❖ to limit the type conversion burden on the IOC.
- ◆ **... request the matching `DBR_CTRL_<type>` *once***
  - ❖ to get the full channel detail (units, limits, ...).
- ◆ **... and then subscribe to `DBR_TIME_<type>` to get updates of only time/status/value**
  - ❖ so now we always stay informed, yet limit the network traffic.
  - ❖ *Only subscribe once*, not with each connection, because CA client library will automatically re-activate subscriptions!
- ◆ **This is what EDM, archiver, ... do.**
  - ❖ Quirk: They don't learn about online changes of channel limits, units, ...  
Doing that via a subscription means more network traffic, and CA doesn't send designated events for 'meta information changed'.



# Simple CA Client

## ◆ Defines and includes

```
/* Simple CA client */
```

```
#define TIMEOUT 1.0
```

```
#define SCA_OK 1
```

```
#define SCA_ERR 0
```

```
#define MAX_STRING 40
```

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

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

```
#include <string.h>
```

```
#include <caodef.h>
```





# Simple CA Client

## ◆ Function prototypes and global variables

```
/* Function prototypes */
int main(int argc, char **argv);
static int parseCommand(int argc, char **argv);
static void usage(void);

/* Global variables */
int pvSpecified=0;
char name[MAX_STRING];
char value[MAX_STRING];
double timeout=TIMEOUT;
```



# Simple CA Client

## ◆ Parse the command line

```
int main(int argc, char **argv)
{
    int stat;
    chid pCh;

    /* Parse the command line */
    if(parseCommand(argc,argv) != SCA_OK)
exit(1);
    if(!pvSpecified) {
        printf("No PV specified\n");
        exit(1);
    }
}
```



# Simple CA Client

## ◆ Initialize Channel Access

```
/* Initialize */
stat=ca_context_create(ca_disable_preemptive_callback);
if(stat != ECA_NORMAL) {
    printf("ca_context_createfailed:\n%s\n",
        ca_message(stat));
    exit(1);
}
```





# Simple CA Client

## ◆ Request the search

```
/* Search */
```

```
stat=ca_create_channel(name, NULL, NULL,  
    CA_PRIORITY_DEFAULT, &pCh) ;  
if(stat != ECA_NORMAL) {  
    printf("ca_create_channel failed:\n%s\n",  
        ca_message(stat)) ;  
    exit(1) ;  
}
```





# Simple CA Client

## ◆ Call ca\_pend\_io to process the search

```
/* Process search */
stat=ca_pend_io(timeout);
if(stat != ECA_NORMAL) {
    printf("search timed out after %g sec\n",
        timeout);
    exit(1);
}
```





# Simple CA Client

## ◆ Request the get

```
/* Get the value */
stat=ca_array_get(DBR_STRING,1,pCh,&value);
if(stat != ECA_NORMAL) {
    printf("ca_array_get:\n%s\n",
        ca_message(stat));
    exit(1);
}
```





# Simple CA Client

## ◆ Call `ca_pend_io` to process the get

```
/* Process get */
stat=ca_pend_io(timeout);
if(stat != ECA_NORMAL) {
    printf("get timed out after %g sec\n",
        timeout);
    exit(1);
}
printf("The value of %s is %s\n",name,value)
```



# Simple CA Client

## ◆ Clean up

```
/* Clear the channel */
stat=ca_clear_channel(pCh);
if(stat != ECA_NORMAL) {
    printf("ca_clear_channel failed:\n%s\n",
        ca_message(stat));
}

/* Exit */
ca_context_destroy();
return(0);

}
```



# SEVCHK

## ◆ For simple error handling in test programs

❖ **SEVCHK ( <function call>, "message" )**

- ❖ **Macro that checks return codes**
- ❖ **If error, displays message and aborts**
- ❖ **Used in example programs**
- ❖ **DON'T use for robust clients**





# Simple CA Client

## ◆ Output

```
simplecaget evans:calc
```

```
The value of evans:calc is 6
```





# Simple CA Client with Callbacks

## ◆ Defines and includes

```
/* Simple CA client with Callbacks */
```

```
#define TIMEOUT 1.0  
#define SCA_OK 1  
#define SCA_ERR 0  
#define MAX_STRING 40
```

```
#include <stdio.h>  
#include <stdlib.h>  
#include <time.h>  
#include <string.h>  
#include <caodef.h>
```





# Simple CA Client with Callbacks

## ◆ Function prototypes

```
/* Function prototypes */  
int main(int argc, char **argv);  
static void connectionChangedCB(struct connection_handler_args args);  
static void valueChangedCB(struct event_handler_args args);  
static char *timeStamp(void);  
static int parseCommand(int argc, char **argv);  
static void usage(void);
```





# Simple CA Client with Callbacks

## ◆ Global variables

```
/* Global variables */  
int pvSpecified=0;  
char name[MAX_STRING];  
time_t curTime, startTime;  
double timeout=TIMEOUT;
```



# Simple CA Client with Callbacks

## ◆ Parse the command line

```
int main(int argc, char **argv)
{
    int stat;
    chid pCh;

    /* Parse the command line */
    if(parseCommand(argc,argv) != SCA_OK) exit(1);
    if(!pvSpecified) {
        printf("No PV specified\n");
        exit(1);
    }
}
```



# Simple CA Client with Callbacks

## ◆ Initialize Channel Access

```
/* Initialize */
stat=ca_context_create(ca_disable_preemptive_callback);
if(stat != ECA_NORMAL) {
    printf("ca_context_createfailed:\n%s\n",
        ca_message(stat));
    exit(1);
}
```



# Simple CA Client with Callbacks

## ◆ Search

```
/* Search */
stat=ca_create_channel(name,connectionChangedCB,NULL,
    CA_PRIORITY_DEFAULT,&pCh);
if(stat != ECA_NORMAL) {
    printf("ca_create_channel failed:\n%s\n",
        ca_message(stat));
    exit(1);
}
printf("%s Search started for %s\n",timeStamp(),name);
```



# Simple CA Client with Callbacks

- ◆ Wait in `ca_pend_event` for the callbacks to occur

```
/* Wait */
startTime=curTime;
ca_pend_event(timeout);
printf("%s ca_pend_event timed out after %g sec\n",
       timeStamp(), timeout);
```



# Simple CA Client with Callbacks

## ◆ Clean up

```
/* Clear the channel */
stat=ca_clear_channel(pCh);
if(stat != ECA_NORMAL) {
    printf("ca_clear_channel failed:\n%s\n",
        ca_message(stat));
}

/* Exit */
ca_context_destroy();
return(0);
}
```



# Simple CA Client with Callbacks

## ◆ Connection callback implementation

```
static void connectionChangedCB(struct
connection_handler_args args)
{
    chid pCh=args.chid;
    int stat;

    /* Branch depending on the state */
    switch(ca_state(pCh)) {
```





# Simple CA Client with Callbacks

## ◆ Connection callback implementation

```
case cs_conn:
    printf("%s Connection successful\n", timeStamp());
    stat=ca_array_get_callback(DBR_STRING,1,pCh,
        valueChangedCB,NULL);
    if(stat != ECA_NORMAL) {
        printf("ca_array_get_callback:\n%s\n",
            ca_message(stat));
        exit(1);
    }
    break;
```



# Simple CA Client with Callbacks

## ◆ Connection callback implementation

```
case cs_never_conn:
    printf("%s Cannot connect\n", timeStamp());
    break;

case cs_prev_conn:
    printf("%s Lost connection\n", timeStamp());
    break;

case cs_closed:
    printf("%s Connection closed\n", timeStamp());
    break;
}
}
```



# Simple CA Client with Callbacks

## ◆ Value changed callback implementation

```
static void valueChangedCB(struct
event_handler_args args)
{
    /* Print the value */
    if(args.status == ECA_NORMAL && args.dbr) {
        printf("%s Value is: %s\n", timeStamp(),
            (char *)args.dbr);
        printf("Elapsed time: %ld sec\n",
            curTime-startTime);
    }
}
```



# Simple CA Client with Callbacks

## ◆ Output

```
simplecagetcb evans:calc  
Sep 14 18:31:55 Search started for evans:calc  
Sep 14 18:31:55 Connection successful  
Sep 14 18:31:55 Value is: 5  
Elapsed time: 0 sec  
Sep 14 18:31:56 ca_pend_event timed out after 1  
sec
```

## ◆ Time for this operation is typically a few ms



## Source files for Simple Get Clients

- ◆ Some of the code that is not related to Channel Access has not been shown
- ◆ All the files necessary to build a project as an EPICS Extension should be available with the presentation
  - ❖ Makefile
  - ❖ Makefile.Host
  - ❖ simplecaget.c
  - ❖ simplecagetcb.c
  - ❖ LICENSE
- ◆ Stored as simpleCA.tar.gz





# EPICS Build System

- ◆ Supports both native and GNU compilers
- ◆ Builds multiple types of components
  - ❖ libraries, executables, headers, scripts, java classes, ...
- ◆ Supports multiple host and target operating systems
- ◆ Builds for all hosts and targets in a single <top> tree
  - ❖ epics/base
  - ❖ epics/extensions
- ◆ Allows sharing of components across <top> trees
- ◆ Has different rules and syntax for 3.13 and 3.14





# System Requirements

## ◆ Required software

- ❖ Perl version 5 or greater
- ❖ GNU make, version 3.78.1/3.81 or greater
- ❖ C++ compiler and linker (GNU or host vendor's compiler)

## ◆ Optional software

- ❖ Tornado II and board support packages
- ❖ RTEMS development tools and libraries
- ❖ Motif, X11, JAVA, Tcl/Tk, Python...





# User Requirements

## ◆ Set an environment variable to specify the architecture

### ❖ EPICS\_HOST\_ARCH for 3.14

✧ solaris-sparc, linux-x86, win32-x86, darwin-ppc, etc.

### ❖ HOST\_ARCH for 3.13

✧ solaris, Linux, WIN32, etc.

## ◆ Set the PATH so the required components can be found

❖ Perl, GNU make, C and C++ compilers

❖ System commands (e.q. cp, rm, mkdir)







# Some Pointers to Documents

## ◆ Example files

- ❖ <http://www.aps.anl.gov/epics/>
- ❖ Documents - Training - Developing Client Tools
  - Introduction to Channel Access Clients
  - Example Files

## ◆ Build examples of EPICS-Base, etc on several Platforms

- ❖ <http://www-linac.kek.jp/cont/epics/win32/>
- ❖ <http://www-linac.kek.jp/cont/epics/linux/>
- ❖ <http://www-linac.kek.jp/cont/epics/darwin/>





## Some Examples of Channel-Access

- ◆ Of course, this Presentation
- ◆ `makeBaseApp.pl -t caClient {app-name}`
  - ❖ `caExample.c`
  - ❖ `caMonitor.c`
- ◆ `makeBaseEx.pl -t example {ext-name}`
- ◆ `caExample.c`



# Typical Extensions Build Tree

<b>epics/base</b>	<b>&lt;top&gt; for base</b>
<b>epics/extensions</b>	<b>&lt;top&gt; for extensions</b>
<b>config</b>	<b>3.13 configuration</b>
<b>configure</b>	<b>3.14 configuration</b>
<b>bin</b>	<b>Binaries by architecture</b>
<b>solaris</b>	
<b>solaris-sparc</b>	
<b>lib</b>	<b>Libraries by architecture</b>
<b>solaris</b>	
<b>solaris-sparc</b>	
<b>src</b>	<b>Sources by application</b>
<b>simpleCA</b>	<b>Application source files</b>
<b>O.solaris</b>	<b>Binaries for this application</b>
<b>O.solaris-sparc</b>	





# Getting Started with an Extension

- ◆ **Make a directory structure for base `http://www.aps.anl.gov/epics/extensions/index.php`**
  - ❖ **E.g. `epics/base`**
- ◆ **Obtain base and build it**
  - ❖ **Set `COMPAT_TOOLS_313` first if necessary (see later)**
- ◆ **Make a directory structure for extensions**
  - ❖ **E.g. `epics/extensions`**
- ◆ **Get `extensions/config` and configure from the EPICS pages**
  
- ◆ **Set `EPICS_BASE` to your desired version of base**
  - ❖ **In `extensions/config/RELEASE` for 3.13**
  - ❖ **In `extensions/configure/RELEASE` for 3.14**
- ◆ **Type `gnumake` (or `make`) in extensions**
- ◆ **Get an extension and put it under `extensions/src`**
- ◆ **Type `gnumake` (or `make`) in your application directory**



## Using the 3.13 Build Rules for Extensions

- ◆ **Most existing extensions are still set up for 3.13 builds**
  - ❖ There is a Makefile and a Makefile.Host
  - ❖ Makefile.Host is most important and has 3.13 syntax
  - ❖ Can still use a 3.14 base
- ◆ **Set HOST\_ARCH for your platform**
  - ❖ solaris, Linux, WIN32, etc.
- ◆ **Set EPICS\_HOST\_ARCH for your platform**
  - ❖ solaris-sparc, linux-x86, win32-x86, darwin-ppc, etc.
- ◆ **Configuration is in extensions/config**
  - ❖ RELEASE (Specifies what base to use, can be 3.14)
  - ❖ CONFIG\_SITE\_xxx (Specifies local changes for xxx arch)
- ◆ **Before building a 3.14 base**
  - ❖ Modify base/configure/CONFIG\_SITE
    - ✧ **COMPAT\_TOOLS\_313 = YES**





## Using the 3.14 Build Rules for Extensions

- ◆ **Go to the the EPICS page for your version of base**
  - ❖ <http://www.aps.anl.gov/epics/base/index.php>
- ◆ **Read the README**
  - ❖ It is very extensive
  - ❖ Should tell you everything you need to know
- ◆ **There is a only a Makefile and it uses 3.14 syntax**
- ◆ **Set EPICS\_HOST\_ARCH for your platform**
  - ❖ solaris-sparc, linux-x86, win32-x86, darwin-ppc, etc.
- ◆ **Configuration is in extensions/configure**
  - ❖ RELEASE (Specifies what base)
  - ❖ os/CONFIG\_SITE\_xxx (Specifies local changes for xxx arch)





# Makefile for Simple Get Clients

```
TOP = ../../..
```

```
include $(TOP)/config/CONFIG_EXTENSIONS
```

```
include $(TOP)/config/RULES_ARCHS
```





# Makefile.Host for Simple Get Clients

```
TOP = ../../..  
include $(TOP)/config/CONFIG_EXTENSIONS  
  
HOST_OPT = NO  
CMPLR = STRICT  
  
PROD = simplecaget simplecagetcb  
  
PROD_LIBS = ca Com  
ca_DIR = $(EPICS_BASE_LIB)  
Com_DIR = $(EPICS_BASE_LIB)  
  
simplecaget_SRCS += simplecaget.c  
simplecagetcb_SRCS += simplecagetcb.c  
  
include $(TOP)/config/RULES.Host
```







# Acknowledgements

- ◆ **Jeff Hill [LANL] is responsible for EPICS Channel Access and has developed almost all of it himself**
- ◆ **Janet Anderson [ANL] is responsible for and has developed most of the EPICS Build System**





# Thank You

# 谢谢您

