Argonne NATIONAL LABORATORY

Channel Access Client Programming

Andrew Johnson — Computer Scientist, AES-SSG

Channel Access

- The main programming interface for writing Channel Access clients is the library that comes with EPICS base
  - Written in C++, the API is pure C
- Almost all CA client APIs for other languages call the C library
  - Main exception: Pure Java library 'CAJ'
- Documentation:
  - *EPICS R3.14 Channel Access Reference Manual* by Jeff Hill et al.
  - Available in <base>/html, or from the EPICS web site
- This lecture covers
  - Fundamental API concepts and routines
  - Data types and usage
  - Template examples

Search and Connect to a PV

```
use lib '/path/to/base/lib/perl';
use CA;

my @access = ('no ', '');

my $chan = CA->new($ARGV[0]);
CA->pend_io(1.0);

printf "PV: %s\n", $chan->name;
printf "  State:         %s\n", $chan->state;
printf "  Server:        %s\n", $chan->host_name;
printf "  Access rights: %sread, %swrite\n",
    $access[$chan->read_access], $access[$chan->write_access];
printf "  Data type:     %s\n", $chan->field_type;
printf "  Element count: %d\n", $chan->element_count;
```

- This is the basic cainfo program in Perl (without friendly error reporting)

Search and Connect in C

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

char *connState[] = {"Never", "Previously", "Connected", "Closed"};
#define access(v) (v ? "" : "no ");

int main(int argc, char **argv) {
    chid chan;

    SEVCHK(ca_create_channel(argv[1], NULL, NULL, 0, &chan),
        "Create channel failed");
    SEVCHK(ca_pend_io(1.0), "CA Search failed");

    printf("PV: %s\n", ca_name(chan));
    printf("  State:         %s\n", connState[ca_state(chan)]);
    printf("  Server:        %s\n", ca_host_name(chan));
    printf("  Access rights: %sread, %swrite\n",
        access(ca_read_access(chan)), access(ca_write_access(chan));
    printf("  Data type:     %s\n", dbr_type_to_text(ca_field_type(chan)));
    printf("  Element count: %u\n", ca_element_count(chan));
}
```

Get and Put a PV

```
use lib '/path/to/base/lib/perl';
use CA;

my $chan = CA->new($ARGV[0]);
CA->pend_io(1.0);

$chan->get;
CA->pend_io(1.0);
printf "Old Value: %s\n", $chan->value;

$chan->put($ARGV[1]);
CA->pend_io(1.0);

$chan->get;
CA->pend_io(1.0);
printf "New Value: %s\n", $chan->value;
```

- This is the basic caput program in Perl (without friendly error reporting)

Get and Put in C

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

int main(int argc, char **argv) {
    chid chan;
    dbr_string_t value;

    SEVCHK(ca_create_channel(argv[1], NULL, NULL, 0, &chan),
        "Create channel failed");
    SEVCHK(ca_pend_io(1.0), "Search failed");

    SEVCHK(ca_get(DBR_STRING, chan, &value), "Get failed");
    SEVCHK(ca_pend_io(1.0), "Pend I/O failed");
    printf("Old Value: %s\n", );

    SEVCHK(ca_put(DBR_STRING, chan, argv[2]), "Put failed");
    SEVCHK(ca_pend_io(1.0), "Pend I/O failed");

    SEVCHK(ca_get(DBR_STRING, chan, &value), "Get failed");
    SEVCHK(ca_pend_io(1.0), "Pend I/O failed");
    printf("New Value: %s\n", );
}
```

Monitor a PV

```
use lib '/path/to/base/lib/perl';
use CA;

my $chan = CA->new($ARGV[0]);
CA->pend_io(1.0);

$chan->create_subscription('v', \&val_callback);
CA->pend_event(0.0);

sub val_callback {
   my ($chan, $status, $data) = @_;
   if (!$status) {
       printf "PV: %s\n", $chan->name;
       printf "  Value: %s\n", $data;
   }
}
```

- This is a basic camonitor program in Perl (without error checking)

Monitor in C

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

void val_callback(struct event_handler_args eha) {
    if (eha.status == ECA_NORMAL) {
        printf("PV: %s\n", ca_name(eha.chid));
        printf("  Value: %s\n", (const char *) eha.dbr);
    }
}

int main(int argc, char **argv) {
    chid chan;

    SEVCHK(ca_create_channel(argv[1], NULL, NULL, 0, &chan),
        "Create channel failed");
    SEVCHK(ca_pend_io(1.0), "Search failed");

    SEVCHK(ca_create_subscription(DBR_STRING, 1, chan, DBE_VALUE,
        val_callback, NULL, NULL),"Subscription failed");
    SEVCHK(ca_pend_event(0.0), "Pend event failed");
}
```

Handling Errors

- What happens if the PV search fails, e.g. the IOC isn't running, or it's busy and takes longer than 1 second to reply?
  - In Perl:
    - `CA->pend_io(1.0)` throws a Perl exception (die)
    - Program exits after printing:

      ECA_TIMEOUT - User specified timeout on IO operation expired at test.pl line 5.
    - We could trap that exception using

```
eval {CA->pend_io(1)};
if ($@ =~ m/^ECA_TIMEOUT/) { ... }
```

  - In C:
    - `ca_pend_io(1.0)` returns ECA_TIMEOUT
    - `SEVCHK()` prints a message and calls abort()

- Problem with these approaches:
  - How to write a program that doesn't require the IOC to be running when it starts up?

Event-driven Programming

- First seen when setting up the CA monitor:

```
$chan->create_subscription('v', \&val_callback);
CA->pend_event(0.0);
```

  - The CA library will run the `val_callback` subroutine whenever the server sends a new data value for this channel
  - The program must be inside a call to `CA->pend_event()` or `CA->pend_io()` for the CA library to execute callback routines
    - Multi-threaded C programs can avoid this requirement (Perl programs can't)
    - Callbacks are executed by other threads created inside the CA library

- Most CA functionality can be event-driven
- It is legal to call most CA routines from within a callback subroutine
  - The main exceptions are `ca_poll()`, `ca_pend_event()` and `ca_pend_io()`

Event-driven PV Search and Connect

```
use lib '/path/to/base/lib/perl';
use CA;

my @chans = map {CA->new($_, \&conn_callback)} @ARGV;
CA->pend_event(0);

sub conn_callback {
    my ($chan, $up) = @_;
    printf "PV: %s\n", $chan->name;
    printf "  State:          %s\n", $chan->state;
    printf "  Host:           %s\n", $chan->host_name;
    my @access = ('no ', '');
    printf "  Access rights: %sread, %swrite\n",
        $access[$chan->read_access], $access[$chan->write_access];
    printf "  Data type:      %s\n", $chan->field_type;
    printf "  Element count: %d\n", $chan->element_count;
}
```

- The cainfo program using callbacks

Event-driven Search and Connect in C

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

char *connState[] = {"Never", "Previously", "Connected", "Closed"};
#define access(v) (v ? "" : "no ");

void conn_callback(struct ca_connection_handler_args cha) {
    printf("PV: %s\n", ca_name(cha.chid));
    printf("  State:         %s\n", connState[ca_state(cha.chid)]);
    printf("  Server:        %s\n", ca_host_name(cha.chid));
    printf("  Access rights: %sread, %swrite\n",
        access(ca_read_access(cha.chid)), access(ca_write_access(cha.chid));
    printf("  Data type:     %s\n", dbr_type_to_text(ca_field_type(cha.chid)));
    printf("  Element count: %u\n", ca_element_count(cha.chid));
}

int main(int argc, char **argv) {
    for (int i = 1; i < argc; i++) {
        chid chan;
        SEVCHK(ca_create_channel(argv[i], conn_callback, NULL, 0, &chan),
            "Create channel failed");
    }
    SEVCHK(ca_pend_event(0.0), "Pend event returned");
}
```

Event-driven PV Monitor

```
use lib '/path/to/base/lib/perl';
use CA;
my @chans = map {CA->new($_, \&conn_cb)} @ARGV;
CA->pend_event(0);
sub conn_cb {
    my ($ch, $up) = @_;
    if ($up && ! $monitor{$ch}) {
        $monitor{$ch} = $ch->create_subscription('v', \&val_cb);
    }
}
sub val_cb {
   my ($ch, $status, $data) = @_;
   if (!$status) {
       printf "PV: %s\n", $ch->name;
       printf "  Value: %s\n", $data;
   }
}
```

- The camonitor program using callbacks

Event-driven Monitor in C

- Student exercise:
  - Write a program in C that
    - Accepts a list of PV names from the command line
    - Connects to these PVs and monitors them for value changes
    - Prints the new values to stdout as they arrive
    - Still works properly after an IOC reboot
  - Look at previous slides, or the CA Reference Manual
  - Don't worry about compiling it yet

Data Types for C code

- CA routines take an integer **`type`** argument to indicate the data type to transfer
- These are macros defined in db_access.h

Name Macro	Data Type	Type Definition	
DBR_CHAR	dbr_char_t	epicsInt8	*any, num*
DBR_SHORT	dbr_short_t	epicsInt16	*any, num*
DBR_LONG	dbr_long_t	epicsInt32	*any, num*
DBR_FLOAT	dbr_float_t	epicsFloat32	*any, num*
DBR_DOUBLE	dbr_double_t	epicsFloat64	*any, num*
DBR_ENUM	dbr_enum_t	epicsUInt16	*any*
DBR_STRING	dbr_string_t	char [40]	*any*
DBR_STS_*any*	struct dbr_sts_*any*	{ alrm, val }	
DBR_TIME_*any*	struct dbr_time_*any*	{ alrm, stamp, val }	
DBR_GR_*num*	struct dbr_gr_*num*	{ alrm, units, disp, val }	
DBR_CTRL_*num*	struct dbr_ctrl_*num*	{ alrm, units, disp, ctrl, val }	
DBR_GR_ENUM	struct dbr_gr_enum	{ alrm, no_str, strs[], val }	
DBR_CTRL_ENUM	struct dbr_ctrl_enum	{ alrm, no_str, strs[], val }	
DBR_PUT_ACKT	dbr_put_ackt_t	epicsUInt16	
DBR_PUT_ACKS	dbr_put_acks_t	epicsUInt16	
DBR_STSACK_STRING	struct dbr_stsack_string	{ alrm, ackt, acks, val }	
DBR_CLASS_NAME	dbr_class_name_t	char [40]	

Excerpt from db_access.h

```
/*
 *   DBR_CTRL_DOUBLE returns a control double structure (dbr_ctrl_double)
 */

/* structure for a control double request */
struct dbr_ctrl_double{
    dbr_short_t     status;                 /* status of value */
    dbr_short_t     severity;               /* severity of alarm */
    dbr_short_t     precision;              /* number of decimal places */
    dbr_short_t     RISC_pad0;              /* RISC alignment */
    char            units[MAX_UNITS_SIZE];  /* units of value */
    dbr_double_t    upper_disp_limit;       /* upper limit of graph */
    dbr_double_t    lower_disp_limit;       /* lower limit of graph */
    dbr_double_t    upper_alarm_limit;
    dbr_double_t    upper_warning_limit;
    dbr_double_t    lower_warning_limit;
    dbr_double_t    lower_alarm_limit;
    dbr_double_t    upper_ctrl_limit;       /* upper control limit */
    dbr_double_t    lower_ctrl_limit;       /* lower control limit */
    dbr_double_t    value;                  /* current value */
};
```

Array Data

- Calls to **`ca_`*`xxx`*`()`** are equivalent to **`ca_array_`*`xxx`*`()`** with a count of 1
- The **`ca_element_count()`** macro gives the maximum possible array size
  - Value is sent by the server just once, at connection time
- Arrays can contain less data; the IOC knows the current array size
  - Before Base release 3.14.12 the CA library would always add zero values after the valid array elements to fill it up to the maximum size (or the size requested)
- From Base 3.14.12 onward, you can pass a count of 0 into **`ca_array_get_callback()`** and **`ca_create_subscription()`** to fetch only the valid array elements
  - The callback is given the number of elements provided
    - This will never be greater than **`ca_element_count()`**
  - For subscription callbacks, that number may be different every time

String Handling

- A **`dbr_string_t`** value (**`DBR_STRING`** field) uses a fixed length 40 character buffer
  - A terminating zero will always be present
  - Some record fields can only hold fewer characters, e.g. EGU
- Longer strings can be stored in a **`dbr_char_t`** array
  - Waveform record type, or some other array field
  - A terminating zero element might not be present
- Newer IOCs also support accessing string fields as a **`DBR_CHAR`** array
  - A terminating zero *should* be present

Specifying Data Types in Perl

- Most of the Perl I/O routines handle the channel data types automatically
  - **`$chan->get`** fetches one element in the channel's native type
    - Value is returned by **`$chan->value`**
    - Arrays are not supported, no type request possible
  - **`$chan->get_callback(SUB)`** fetches all elements in the channel's native data type
    - Optional TYPE and COUNT arguments to override
  - **`$chan->create_subscription(MASK, SUB)`** requests all elements in the channel's native type
    - Optional TYPE and COUNT arguments to override
  - **`$chan->put(VALUE)`** puts values in the channel's native type
    - VALUE may be a scalar or an array
  - **`$chan->put_callback(SUB, VALUE)`** puts values in the channel's native data type
    - VALUE may be a scalar or an array

Perl Data Type Parameters

- The TYPE argument is a string naming the desired **`DBR_xxx`** type
- The COUNT argument is the integer number of elements

- If the data contains multiple elements, the callback subroutine's **`$data`** argument becomes an array reference
- If the data represents a composite type, the callback subroutine's **`$data`** argument becomes a hash reference
  - The hash elements included are specific to the type requested
  - See the Perl CA Library documentation for more details

Multi-threading

- The CA client library is thread-aware
  - Can be used in both single- and multi-threaded environments
  - Uses threads internally, 2 per server it connects to
  - Callbacks are usually executed by one of the server-specific threads
- Applications can configure callbacks to be run preemptively
  - By default, callbacks are only run when the application is inside `ca_pend_io()`, `ca_poll()` or `ca_pend_event()`
  - Call `ca_context_create(ca_enable_preemptive_callback);` to change that
  - The application is then responsible for using mutexes to protect shared resources etc.
- Use `ca_current_context()` and `ca_attach_context()` to share a single CA client context between multiple application threads

Ideal CA client?

- Register and use callbacks for everything
  - Event-driven programming; polling loops or fixed time outs
  - On connection, check the channel's native type
    - Limit the data type conversion burden on the IOC
  - Subscribe for DBE_PROPERTY updates using the **`DBR_CTRL_`*`type`***
    - This provides the full channel detail (units, limits, …)
    - Future IOCs will send property events when those attributes change
  - Subscribe for value updates using **`DBR_TIME_`*`type`*** to get time+alarm+value
  - Only subscribe once at first connection; the CA library automatically re-activates subscriptions after a disconnect/reconnect
    - However, be prepared in case the channel's native type changes (rare, but this can happen)
- This gives updates without having to poll for changes

Quick Hacks, Scripts

- In many cases, scripts written in bash/perl/python/php can just invoke the command-line ‘caget’ and ‘caput’ programs
- Especially useful if you only need to read/write one PV value and not subscribe to value updates
- CA Client library bindings are available for Perl, Python & PHP
  - Perl bindings are included in EPICS Base (not on MS Windows)
  - You have to find, build and update them for Python and PHP
    - Your script may be portable, but you still have to install the CAC-for-p* binding separately for Linux, Win32, MacOS…

Base caClient template

- EPICS Base Includes a makeBaseApp.pl template that builds two basic CA client programs written in C:
  - Run this:

```
mkdir client && cd client
../base/bin/darwin-x86/makeBaseApp.pl -t caClient cacApp
make
```

  - Builds two programs:

```
bin/darwin-x86/caExample pvName
bin/darwin-x86/caMonitor pvListFile
```

caClient Example Programs

- caExample.c
  - Minimal CA client program
  - Fixed timeout, waits until data arrives
  - Requests everything as **`DBR_DOUBLE`**
- caMonitor.c
  - Better CA client program
  - Registers callbacks for connections, exceptions, access rights
  - Subscribes for value updates
    - Only uses one data type (**`DBR_STRING`**) for everything