CA Client Programming in Perl and C

Andrew Johnson — AES/SSG, Argonne

Includes material from:

Ken Evans, Argonne

Kay Kasemir, ORNL

Task: Write a Channel Access client

- Many possible approaches and choices of language
- Assuming that you need more than you can do with
  - MEDM/EDM/CaQtDm/EpicsQt display manager
  - CSS/Boy with its rules and scripts
- These are commonly used options
  - Shell or Perl script that calls the caget/caput/camonitor programs
  - Python program with PyEpics or EPICS Cothread bindings
  - Matlab/Octave/Scilab with MCA or LabCA bindings
  - State Notation Language (SNL) program with the Sequencer
  - Perl program with CA bindings
  - C++ program with EPICS Qt bindings
  - Java program calling CAJ (pure Java) or JCA (JNI)
  - C/C++ program calling CA library

SNL programs speak CA natively

- This piece of SNL handles all the connection management and data type handling:

  -
    ```
    double value;
    assign value to "fred";
    monitor value;
    ```
- Extend into a basic 'camonitor':

  -
    ```
    evflag changed;
    sync value changed;

    ss monitor_pv
    {
       state check
       {
           when (efTestAndClear(changed))
           {
               printf("Value is now %g\n", value);
           } state check
       }
    }
    ```

Quick Hacks, Simple Scripts

- In many cases, scripts written in bash/perl/python/php can just invoke the command-line ‘caget’ and ‘caput’ programs
- Useful for reading/writing one or two PV values, not for subscribing to value updates
- Quiz: Why would a loop that continually invokes ‘caget’ or ‘caput’ be bad?

- CA Client library bindings are available for Perl, Python & PHP
  - Perl bindings are included in EPICS Base (not available on MS Windows)
  - Several different Python bindings are available
- Much better to use these for long-running scripts

Simple Script Example

```perl
#!/bin/env perl -w

# caget: Get the current value of a PV
# Argument: PV name
# Result: PV value
sub caget {
    my ($pv) = @_;
    open(my $F, "-|", "caget -t $pv") or die "Cannot run 'caget'\n";
    $result = <$F>;
    close $F;
    chomp $result;
    return $result;
}

# Do stuff with PVs
my $fred = caget("fred");
my $jane = caget("jane");
my $sum = $fred + $jane;
printf("Sum: %g\n", $sum);
```

Channel Access for Perl, C and C++

- The Channel Access client library comes with EPICS base and is the basis for most of the other language bindings
  - Internally written in C++ but API is pure C
  - Main exception: Pure Java library ‘CAJ’
- Documentation:
  - *EPICS R3.14 Channel Access Reference Manual* by Jeff Hill et al.
  - *CA - Perl 5 interface to EPICS Channel Access* by Andrew Johnson
  - In <base>/html, or from the EPICS web site
- This section covers
  - Fundamental API concepts using Perl examples
  - Some brief examples in C
  - How to instantiate a template with some example C programs

CA Client APIs for Perl, C and C++

- Why teach the Perl API before C?
  - Higher level language than C, no pointers needed
  - Learn the main principles and library calls with less code
  - Complete Perl programs can fit on one slide

- The Perl 5 API is a thin wrapper around the C library
  - Built with Base on most Unix-like workstation platforms (not Windows)
  - Provides the same interface model that C code uses
  - Unless you're interfacing to specific libraries or need very high performance, Perl scripts may be sufficient for most tasks

- Other APIs like Python and Java are less like the C library
  - Good for writing client programs in Python/Java, but not for learning the C library

Search and Connect to a PV

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

my $chan = CA->new($ARGV[0]);
CA->pend_io(1);

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;
```

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

Get and Put a PV

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

my $chan = CA->new($ARGV[0]);
CA->pend_io(1);

$chan->get;
CA->pend_io(1);
printf "Old Value: %s\n", $chan->value;

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

$chan->get;
CA->pend_io(1);
printf "New Value: %s\n", $chan->value;
```

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

Monitor a PV

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

my $chan = CA->new($ARGV[0]);
CA->pend_io(1);

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

Error Checking

- 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?
  - `CA->pend_io(1)` times out
  - CA library throws a Perl exception (die)
  - Program exits after printing:
    - ECA_TIMEOUT - User specified timeout on IO operation expired at test.pl line 5.
- We can trap the Perl exception using

  -
    ```
    eval {CA->pend_io(1)};
    if ($@ =~ m/^ECA_TIMEOUT/) { ... }
    ```
- How can we write code that can recover from failed searches and continue doing useful work?

Event-driven Programming

- First seen when setting up the CA monitor:
  - **`$chan->create_subscription('v', \&callback);`**
    `CA->pend_event(0);`
  - The CA library executes our callback subroutine whenever the server provides a new data value for this channel
  - The **`CA->pend_event()`** routine must be running for the library to execute callback routines
    - The Perl CA library is single threaded
    - Multi-threaded C programs can avoid this requirement

- Most CA functionality can be event-driven

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 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

Data Type Requests

- Most data I/O routines handle data type 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

Specifying Data Types

- The TYPE argument is a string naming the desired DBR_xxx type
  - See the CA Reference Manual for a list
- The COUNT argument is the integer number of elements

- If you request an array, the callback subroutine's **`$data`** argument becomes an array reference
- If you request a composite type, the callback subroutine's **`$data`** argument becomes a hash reference
  - The hash elements are different according to the type you request
  - See the Perl Library documentation for details

Simple Channel Access calls from C

- Main header file
  - `#include <cadef.h>`
  - This also includes db_access.h, caerr.h and caeventmask.h

- Channels are referred to using as a **`chid`**, a pointer to an opaque structure
  - `chid fred;`

- Connect to a channel
  -
    ```
    int status = ca_create_channel("fred", NULL, NULL, 0, &fred);
    SEVCHK(status, "Create channel failed");
    status = ca_pend_io(1.0);
    SEVCHK(status, "Channel connection failed")
    ```

- The **`SEVCHK(status, text)`** macro is useful for simple programs
  - Aborts with an error message on bad status

What’s in a chid?

- We can get channel information from a connected chid

  -
    ```
    const char *ca_state_to_text[4] = {"Never connected",
        "Not connected", "Connected", "Closed"};

    printf("PV: %s\n", ca_name(fred));
    printf("State: %s\n", ca_state_to_text[ca_state(fred)]);
    printf("Host:  %s\n", ca_host_name(fred));
    printf("Read:  %s\n", ca_read_access(fred) ? "Y" : "N");
    printf("Write: %s\n", ca_write_access(fred) ? "Y" : "N");
    printf("Type:  %s\n", dbr_type_to_text(ca_field_type(fred)));
    printf("Count: %s\n", ca_element_count(fred));
    ```

- Tidy up after we’re finished with fred
  - `SEVCHK(ca_clear_channel(fred), "Clear channel failed");`

Writing to a PV

- Assuming the chid fred is already/still connected

  -
    ```
    SEVCHK(ca_put(DBR_STRING, fred, "10"), "Put failed");
    ca_flush_io();
    ```
- If fred's PV can hold an array of doubles

  -
    ```
    dbr_double_t data[] = {1.0, 2.0, 3.0, 4.0, 5.0};

    SEVCHK(ca_array_put(DBR_DOUBLE, 5, fred, data), "Put failed");
    ca_flush_io();
    ```
- What other data types are available?
  - See the db_access.h file in Base/include

Reading from a PV

- Still assuming fred is connected

  -
    ```
    struct dbr_time_double val;
    const char * severity_to_text[4] = {
        "No alarm", "Minor", "Major", "Invalid"};

    SEVCHK(ca_get(DBR_TIME_DOUBLE, fred, &val), "Get failed");
    SEVCHK(ca_pend_io(1.0), "I/O failed");
    printf("PV: %s\n", ca_name(fred));
    printf("value:    %g\n", val.value);
    printf("severity: %s\n", severity_to_text[val.severity]);
    printf("status:   %hd\n", val.status);
    ```

Base caClient template

- EPICS Base Includes a makeBaseApp.pl template that builds two basic CA client programs written in C
  - Type these commands:
    ```
    mkdir clients; cd clients
    makeBaseApp.pl -t caClient clientApp
    make
    ```
  - Try running the result like this:
    ```
    bin/linux-x86/caExample id01:shutter
    ```
    ```
    echo id01:shutter > pvfile
    bin/linux-x86/caMonitor pvfile
    ```
  - Then read the source files in your **`clientApp`** directory, compare with the reference manual, and edit/extend to suit your needs

CaClient's caExample.c

- Minimal CA client program
- Fixed timeout, waits until data arrives
- Requests everything as ‘DBR_DOUBLE’
  - … which results in values of type ‘double’
  - See db_access.h header file for all the DBR_… constants and the resulting C types and 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

Excerpt from db_access.h

```
/* values returned for each field type
 …
 *      DBR_DOUBLE      returns a double precision floating point number
 …
 *      DBR_CTRL_DOUBLE returns a control double structure (dbr_ctrl_double)
 */

…

/* structure for a control double field */
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 */
};
```

caClient'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 one data type (DBR_STRING) for everything

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
- Request the matching DBR_CTRL_*type* once
  - this gets the full channel detail (units, limits, …)
- Then subscribe to DBR_TIME_*type* for time+status+value updates
  - Now we always stay informed, yet limit the network traffic
  - Only subscribe once at first connection; the CA library automatically re-activates subscriptions after a disconnect/reconnect
- This is what CSS, EDM, ALH etc. do
  - Quirk: Most don't learn about run-time changes of limits, units, etc.
    - Recent versions of CA support DBE_PROPERTY monitor event type
    - This will solve that issue, once the programs and gateway use it