Introduction to CDEV

Overview of the Control Device Interface

Chip Watson, Jie Chen, Danjin Wu, Walt Akers

Version 1.5 December 9, 1996

TJNAF - Thomas Jefferson National Accelerator Facility

DOCUMENT DATE: Table of ontents generated: December 9, 1996 11:45 am

TRADEMARKS:

UNIX is a registered trademark of AT&T in the USA and other countries.

VxWorks is a register trademark of Wind River Systems.

The X Window System is a trademark of Massachusetts Institute of Technology.

OSF/Motif and Motif are trademarks of Open Software Foundation, Inc.

Ultrix and DEC are registered trademarks of Digital Equipment Corporation.

HPUX is a registered trademark of Hewlett Packard.

SURA/CEBAF: The Southeastern Universities Research Association (SURA) operates the Continuous Electron Beam Accelerator Facility (CEBAF) for the United States Department of Energy under contract DE-AC05-84ER40150.

Table of Contents

List of Figures

1. Introduction

What is CDEV

The CDEV (common device) C++ class library is an object-oriented framework that provides a standard interface between an application and one or more underlying control packages or systems. Each underlying package (called a service) is dynamically loaded at run-time, allowing an application to access a new service by name without re-compilation or re-linking. A major feature of CDEV is its ability to hide the implementation details of a particular device from the high level application developer. This allows server implementation choices to be changed without breaking high level client applications. Additionally, because the application is unaware of the underlying service's specific implementation, CDEV is a powerful vehicle for developing portable applications.

The basic CDEV interface can be described through a *device*/*message* paradigm. The *device* is not necessarily a physical object, rather, it is a cdevDevice object that provides a standard interface to CDEV. The application uses this interface to send messages to the device. When the cdevDevice receives this message it will dynamically load the appropriate service (as defined in the CDEV device definition file), and forward the message to it. The service may then perform any action necessary to satisfy the application's request.

The CDEV API is optimized for repetitive operations, in particular for monitoring value changes within a server. Asychronous operations with callbacks are supported, including asychronous error notification. Specifically, CDEV supports most features of channel access (the EPICS communications protocol).

Layering and Abstraction

The CDEV library provides a standard interface to one or more underlying control packages or systems. Currently CDEV provides support for EPICS channel access, tcl scripts, the ACE communications package (a C++ class library over TCP/IP). The ACE package is used to communicate with an on-line accelerator modeling package and with CODA, the CEBAF On-line Data Acquisition system. Access to the EPICS archive data will be added in a later release. All data will be available through a single API (application programming interface).

Information Hiding

The ability to hide the implementation details of a particular device (such as the choice of records in EPICS) from the high level application developer, allows server implementation choices to be changed without breaking all high level client applications. For example, several records may be merged into a new custom record without changing applications which access those records through the CDEV API.

Device Object

The basic idea behind CDEV is that all I/O is performed by sending messages to devices. A device is a named entity in the control system which can respond to a set of messages such as *on* or *off* or *get current*. Further, a device is a virtual entity potentially spanning multiple servers and services such as channel access, an archiver, and a static database.

Each device and message pair is mapped to a unique service (such as channel access) and service dependent data (such as the process variable name in EPICS). This mapping is kept in a name service, and is ultimately initialized from an ASCII file. The designer/maintainer of the server would typically be the person responsible for keeping the mapping current.

Standard Messages

Most devices will implement a number of attributes which may be read or written. In order to facilitate developing generic applications, the following standard messages are recommended for devices, where *attrib* is some attribute of the device:

```
set attrib            => Set the value of the attribute
get attrib            => Get the value of the attribute
monitorOn attrib      => Start monitoring the value of the attribute
monitorOff attrib     => Stop monitoring the value of the attribute
```

Data Object

The message sent to the device may have an associated data object. For example, the message *set current* will have an associated *value* for the new current setting. This value is passed as an additional argument on the send call.

Data returned from the server to the client may contain multiple tagged values, including *value*, *status*, and *timestamp*. Each of these items is extracted from the returned data object by specifying a name or an integer tag. Any data object may contain an arbitrary number of tagged entries, allowing considerable flexibility in defining future clients and servers.

Acknowledgment: This work derives some of its ideas about devices and messages from work done at Argonne by Claude Saunders on a device oriented layer above channel access. Much additional analysis & design work has been done by members of the EPICS collaboration.

2. Basic Operations on Devices and Data

The basic user deals only with the device and data objects. The device is created by name, and I/O consists of sending messages to this device. The device object is the simplest I/O object to use: a single device (magnet, viewer, etc.) may be completely controlled from a single device object, with data being passed by 2 data objects (one for output, another for returned results; dataless commands would not require either). The following sample code reads the magnet current from one magnet, and sets the magnet current for a second magnet to that value:

Figure 1: Sample usage of the send method of a cdevDevice

```
void main ()
    {
    // ***********************************************************
    // * Get references to the 2 devices
    // ***********************************************************
    cdevDevice& mag1 = cdevDevice::attachRef ("Magnet1");
    cdevDevice& mag2 = cdevDevice::attachRef ("Magnet2");
    cdevData result;

    // ***********************************************************
    // * Get the current from the first magnet and use it to set
    // * the current on the second.
    // ***********************************************************
    mag1.send("get current", NULL, result);
    mag2.send("set current", result, NULL);
    }
```

The send call, which is synchronous, takes 3 arguments, and returns an integer status:

```
int send(char* msg, cdevData& out, cdevData& result);
```

Like all CDEV calls, a return code of 0 means success (error handling is described in a later section). The first argument to *send* is a character string message. The device name and this string uniquely determine the underlying service to use, and whatever addressing data that service needs to locate the server. The first *send* operation automatically connects to the requested service, initializing any underlying packages as needed.

The second and third arguments are outbound and result data of type cdevData. This is a composite data object which may contain an arbitrary number of data values tagged with an integer tag. The data is self-describing to the extent that cdevData keeps track of the data type, and can automatically convert between data types through C++ function overloading or by explicit direction from the caller. The integer tags are not known to the user at compile time (i.e., no header files required), and can be referred to by character string equivalents. The cdevData class keeps track of all integer tags and their character string equivalents. Character string tags may be converted to integer tags at run time to improve performance. The following code demonstrates how to insert values into and extract values from a cdevData object:

Figure 2: Overview of cdevData operations

```
cdevData d1, d2;
int valtag;
float x, x2;
int status;
...
d1.tagC2I("value",&valtag);      // get the integer tag for "value"
...
d1.insert("value",37.9);         // insert using a character tag
device.send("message",d1,d2);    // send d1, get back d2
d2.get(valtag,&x);               // get "value" from d2 into x
d2.get("status",&status);        // extract the "status"
x2 = d2;                         // get "value" into x2 (= overload)
```

The last line above demonstrates using operator overloading to extract the "value" item from the data object; i.e., the character tag "value" is the default tag to use in extracting a data item for the "=" function overload. This form is available for all scalar types supported by cdevData.

Most I/O operations use the "value" tag to transmit a single value (scalar or array). Other commonly used tags are "status" and "time" (a POSIX time struct). These tagged data items are optionally returned by some servers under the control of the I/O context (see below for changing the default context).

In this release, cdevData supports the following data types: *unsigned char, short, unsigned short, int, unsigned int, long, unsigned long, float, double*, *character strings* and *time stamp.* All data types may be either scalar or multi-dimensional arrays.

3. Asychronous Operations

In addition to the synchronous send, there are 2 asynchronous forms for sending messages. The class definition for all three forms is given below.

Figure 3: Three forms of the send command

```
class cdevDevice:
{
public:
    int send (char* msg, cdevData& out, cdevData& result);
    int sendNoBlock (char* msg, cdevData& out, cdevData& result);
    int sendCallback (char* msg, cdevData& out, cdevCallback cb);
};
```

The second form, sendNoBlock, takes the same arguments as send but completes asynchronously. That is, the result will be invalid until some synchronizing action is taken (discussed below). The third form returns the result to a callback function specified as part of the callback argument. For each of these asychronous calls, the message may be buffered by the underlying service, so that upon return from the call there is no guarantee that the message has even been sent yet. Transmission may be forced with a flush call using the cdevSystem object described below. A flush is automatically performed if pend is called.

Callback Object

The callback object is a simple object containing 2 items: a function pointer, and an arbitrary user argument. As an example, the following code fragment demonstrates monitoring the magnet current asynchronously (the third argument to the callback function is discussed later):

Figure 4: Overview of the cdevCallback object and the cdevCallbackFunction

```
// declare the callback func
cdevCallbackFunction gotit;

// create callback object
cdevCallback callbk(gotit, myarg);
...
mag1.sendCallback(“monitorOn current”,NULL,callbk);
...

void gotit(int status, void* userarg, cdevRequestObject& reqobj,
          cdevData& result)
    {
    float f = result;
    printf(“new value is %f\n”,f);
    }
```

The following code defines the callback (the cdevRequestObject is described below):

Figure 5: Structure of the cdevCallback object

```
typedef void (*cdevCallbackFunction)(int status, void* userarg,
                                     cdevRequestObject& reqobj,
                                     cdevData& result);
class cdevCallback
{
public:
   cdevCallback (cdevCallbackFunction func, void* userarg);
   cdevCallbackFunction function;
   void* userarg;
}
```

System Object

Waiting for completion of asychronous operations may be accomplished in one of two ways: (1) using groups, and (2) using the system object. The system object keeps track of all devices, and contains methods for flushing, polling, and pending for asychronous operations.

Figure 6: Usage of the pend method in the cdevSystem object

```
// get default system
cdevSystem& sys = cdevSystem::defaultSystem();
...
mag1.sendNoBlock("on",...);  // async op(s)
mag2.sendNoBlock("on",...);  // async op(s)
...
sys.pend();                  // wait for all I/O to complete
```

The following class interface shows the forms for each synchronization method:

Figure 7: Synchronization methods in the cdevSystem object

```
class cdevSystem
{
public:
   int flush();
   int poll();
   int pend();
}
```

flush: Flush all pending output to the network for those services that perform send buffering.

poll: Flush all pending output, and process all received replies (including dispatching callbacks, if any). Any received monitor callbacks will be delivered as well.

pend: Flush all pending output, and process all replies for the specified number of seconds. If the time argument is omitted, wait until all replies have been received. If a monitor operation was started, this waits for the connection (first callback) only.

Group Object

An alternative mechanism for waiting for asychronous calls is through the use of groups. A group keeps track off all I/O operations started from the time the group is started until the group is ended. Groups may be nested or overlapped, and support the same flush/poll/pend operations as the system object. For example:

Figure 8: Sample usage of the cdevGroup object

```
cdevGroup g1, g2;
...
g1.start();
mag1.sendCallback(...);
mag2.sendCallback(...);
g2.start();
bpm1.sendNoBlock(...);
bpm2.sendNoBlock(...);
g1.end();
mag3.sendCallback(...);
bpm3.sendNoBlock(...);
g2.end();
g1.pend(4.0);
// at this point operations on mag1, mag2, bpm1, & bpm2 are done
// or have timed out (error handling discussed later).
...
// mag3 and bpm3 may not be finished yet, pend on the second
// group to wait for their completion
g2.pend();
```

Note: Allowing nested or overlapped groups allows library calls to start or stop groups without needing to know if another group is already active.

Groups may be operated in one of two modes. Immediate mode (default) causes grouped operations to be immediately executed, and the group is just used for completion synchronization. Deferred mode causes messages to be held back until the group is ended and the operations are flushed. After the operations are complete, he group of operations may be flushed again (without re-posting), allowing the group to function as a list manager/executor.

Request Object

Occasionally, an I/O operation may need to be repeated may times. In this case, it is not efficient to parse the message each time to determine which server to use. It is possible to bypass this parsing by creating a request object. The request object is like the device object, except that the message string is specified at creation time, and the request object is therefore specific to the particular underlying service to which that message must be directed.

In the case of EPICS channel access, the request object opens and maintains the channel to the EPICS process variable. The request object is NOT the same as an EPICS channel, in that the request object binds a device and a message, and the

message implies a direction. If the message is considered to be of the form *verb* + *attribute* then the EPICS channel essentially binds device and attribute but not verb. In some future release of CDEV, a more channel like object may be included if there is a demand for it. If two cdevRequestObjects reference the same EPICS channel, only a single channel access connection is established.

The following code demonstrates sending a set of data values to a single magnet using a request object. Note that the send call omits the message argument:

Figure 9: Sample usage of the cdevRequestObject object

```
cdevRequestObject& mag1BDL =
    cdevRequestObject::attachRef("mag1","set BDL");
cdevData mydata;
...
for (i=0;i<100;i++)
    {
    mydata = i*10;              // 10 amp steps
    mag1BDL.send(mydata,NULL);  // ignore errors for now
    sleep(1);
    }
```

In most cases, the device object will create the request object to perform a requested I/O operation. All 3 forms of send are supported by the request object, with a calling syntax identical to the device calls without the message argument. The interface to the request object is given below:

Figure 10: Structure of the cdevRequestObject object

```
class cdevRequestObject
{
public:
    char* message();            // returns the message string
    cdevDevice& device();       // returns the device object
    int state();                // connected/disconnected/
    int access();               // read/write/none
    int send(cdevData& out, cdevData& result);
    int sendNoBlock(cdevData& out, cdevData& result);
    int sendCallback(cdevData& out, cdevCallback callback);
}
```

When an asynchronous operation completes, the relevant request object is passed to the user's callback routine. This allows the user to extract the message string and device name from the request object if necessary or desired.

File Descriptors and "select"

An alternative mechanism for dealing with asynchronous operations is to directly test the file descriptors being used. This interface is currently implemented in the cdevSystem class. The pertinent methods have the following form:

Figure 11: File descriptor routines of the cdevSystem object

```
typedef void (*cdevFdChangedCallback)(int fd, int opened,
                void* userarg)
class cdevSystem
{
public:
    ...
    int getFd(int fd[], int &numFD);
    int addFdChangedCallback(cdevFdChangedCallback cbk, void* arg);
}
```

The getFd method will populate a user allocated array of integers (fd) with the file descriptors currently in use by the cdevSystem object. The number of file descriptors that were allocated by the user is specified in the numFD variable. Upon completion, the numFD variable will be set to the number of file descriptors actually copied into the fd array. An error is returned if the buffer is not sufficient to store the complete set of file descriptors.

The addFdChangedCallback method allows the user to specify a function to be called each time a file descriptor is added (opened=1) or closed (opened=0).

The following code illustrates the usage of the UNIX select call:

Figure 12: Using CDEV file descriptors with the UNIX select function

```
int myFD[5];

void mySelectLoop ( cdevSystem & system )
    {
    int fd[20];
    int numFD = 15, nfds;
    fd_set rfds; // Ready file descriptors
    fd_set afds; // Active file descriptors

    // Copy the file descriptors I am already using to the list
    memcpy(fd, myFD, sizeof(myFD));

    // Get the file descriptors in use by the cdevSystem object
    system.getFd(&fd[5], numFD);

    // Add in the 5 previously defined file descriptors
    numFD += 5;

    // Zero the active file descriptors
    FD_ZERO(&afds);

    // Setup the active file descriptors
    for(int i=0; i<numFD; i++) FD_SET(fd[i], &afds);

    // Get the maximum number of file descriptors
    nfds = FD_SETSIZE;

    while(1) {
        // Copy the active descriptors to the ready descriptors
        memcpy((void *)&rfds, (void *)&afds, sizeof(rfds));

        // Use select to detect activity on the file descriptors
        if(select(nfds, &rfds, 0, 0, 0)<0) exit(1);

        // Iterate through the list of file descriptors
        for(i=0; i<numFD, i++)
            {
            if (FD_ISSET(fd[i], &rfds)
                {
                // Respond to active file descriptor here
                }
            }
        }
    }
```

4. Error Handling

Most CDEV routines return an integer status, with 0=success. Errors may also be automatically printed or routed to a user supplied error handler. Control over this behavior is through the system object:

Figure 13: Error handling mechanisms provided by the cdevSystem object

```
typedef void (*cdevErrorHandler)(int severity, char* text,
                                 cdevRequestObject& request);
class cdevSystem
{
public:
    ...
    int autoErrorOn();
    int autoErrorOff();
    cdevErrorHandler setErrorHandler(cdevErrorHandler handler = 0);
    int reportError(int severity, char *name,
                    cdevRequestObject* request, char* format, ...);
    void setThreshold ( int errorThreshold );
}
```

The *severity* parameter indicates the level of the error. Severity levels are informative, warning, error, and severe error. The following values are defined in cdevErrCode.h to describe the severity of an error message:

Figure 14: Enumerated severity codes generated by CDEV

CDEV_SEVERITY_INFO	**Informative message**
CDEV_SEVERITY_WARN	**Warning message -** operation encountered a problem during processing.
CDEV_SEVERITY_ERROR	**Error message -** operation cannot be completed
CDEV_SEVERITY_SEVERE	**Severe/Fatal Error -** cdev cannot continue execution

autoErrorOn Turn on default error handling, which prints error to stdout.

autoErrorOff Turn off default error handling.

setErrorHandler Set a new function to be the system error handler. Return the old function pointer. Omitting the argument resets the handler to the default handler which simply prints error messages to stderr.

reportError Routine which behaves like printf with 3 additional arguments (severity, caller’s name, and request object if available).

Note: For simplicity of use, a very compact set of error codes is implemented in CDEV. The following error codes are defined in the header file cdevErrCode.h.

Figure 15: Enumerated error codes generated by CDEV

```
CDEV_WARNING         = -2  Failure of function is non-consequential
CDEV_ERROR           = -1  Errors that are not in any categories
CDEV_SUCCESS         = 0   Success
CDEV_INVALIDOBJ      = 1   Invalid cdev objects
CDEV_INVALIDARG      = 2   Invalid argument passed to cdev calls
CDEV_INVALIDSVC      = 3   Wrong service during dynamic loading
CDEV_INVALIDOP       = 4   Operation is unsupported (collection)
CDEV_NOTCONNECTED    = 5   Not connected to low network service
CDEV_IOFAILED        = 6   Low level network service IO failed
CDEV_CONFLICT        = 7   Conflicts of data types or tags
CDEV_NOTFOUND        = 8   Cannot find user request (cdevData)
CDEV_TIMEOUT         = 9   Time out
CDEV_CONVERT         = 10  cdevData Conversion error
CDEV_OUTOFRANGE      = 11  Value out of range for device attribute
CDEV_NOACCESS        = 12  Insufficient access to perform request
CDEV_ACCESSCHANGED   = 13  Change in access permission of device
CDEV_DISCONNECTED    = 60  Application has lost server connection
CDEV_RECONNECTED     = 61  Application has regained server connection
```

5. Name Resolution

Some type of name resolution system is needed to (1) locate which package (called a service) underneath CDEV will support the specified message for the specified device and (2) provide parameters needed by the service to contact the server and perform the desired operation. For example, the message “get current” sent to a magnet “mag1” might use the service “ca” (channel access), with parameter “mag1cur.val” (record and field). This parameter is referred to as service data, i.e. data used by the service to perform the operation. The user is typically unaware of this data, which should be supplied by the device implementer.

cdevDirectory Object

Name resolution in CDEV is implemented by the *cdevDirectory* device. The cdevDirectory device supports a *query* message allowing the user to search the Device Directory List (DDL) for devices which are (1) members of a user specified class or (2) match a user specified regular expression.

As with any cdevDevice object, the user instantiates a cdevDirectory device by using the *cdevDevice::attachPtr()* or *cdevDevice::attachRef()* method. The following code fragment illustrates the correct method for attaching to a cdevDirectory device.

Figure 16: Obtaining a reference to the cdevDirectory device

```
cdevDevice & device = cdevDevice::attachRef(“cdevDirectory”);
```

Like all other devices, the cdevDirectory response to messages. The following messages may be submitted to this device:

- query: Identify the devices that match the selection criteria.
- queryClass: Identify the DDL class from which a device is instantiated.
- queryAttributes: Identify all attributes supported by a device or a DDL class.
- queryMessages: Identify all messages supported by a device or DDL class.
- queryVerbs: Identify all verbs supported by a device or DDL class.
- service: Identify the service that is used by a device/message pair.
- serviceData: Identify service data specified for a device/message pair.
- update: Add information to the cdevDirectory data structure.
- validate: Verify that a device or DDL class contains certain definitions.

By default, there is a single directory device. To allow for arbitrary expansion CDEV will allow each service to register additional directory devices with the system object. A search list of directories may be specified to the system directory device (planned feature).

Each directory acts as if it has a table of information in the following form:

Figure 17: Virtual form of the data in a cdevDirectory

dev-class	dev-name	message	service	service-data
magnet	mag1	on	ca	{pv=M1CSR.VAL default=1}

This table is loaded from a device definition file which describes the mapping from (device,message) to (service,serviceData), where “service” is one of the dynamically loaded CDEV services, and “serviceData” is whatever information that service needs to send the message to the device. In this example the service data contains an EPICS process variable name and a default value to write to that channel.

It is NOT the purpose of the device definition file to completely specify the datatypes of all data passed between device and user. CDEV does considerable data type conversion as needed, and services do reasonable run-time validation. The goal is to make the file as small as is reasonably possible and still maintain readability.

General:

- leading whitespace is not significant
- any amount of whitespace may separate language elements, including space, tab, and newline, with the exception that at least one whitespace character is needed before the colon separating a class name and its inherited classes.
- keywords are case sensitive

NOTE: This specification is preliminary, and will be expanded to support additional features as time allows.

The interface definition consists of three parts: service definitions, device class definitions, and device instantiation (in that order). Standard definitions may be specified via include files using a cpp syntax:

Figure 18: Syntax of the #include directive

```
#include “filename”
```

Service Definition

The service definition declares a service name, and lists all tags which the service will accept for its “serviceData”. The following 3 lines define a service named “myservice” which supports serviceData that specifies values for any of the tags “abc”, “def”, and “ghi”:

Figure 19: Sample service definition

```
service myservice
    {
    tags {abc, def, ghi}
    }
```

As a more concrete example, the following is the specification for the channel access service:

Figure 20: Sample service definition

```
service ca
    {
    tags {pv, default}
    }
```

The “pv” tag is used to specify the process variable name, and the “default” tag is used to specify a default value for a “set” operation (examples are below).

Device Class Definition

Device classes are used to define a collection of similar devices; that is, devices which will respond to the same set of messages. In order to simplify class definition, the following design choices are made:

- multiple inheritance is supported, and
- messages may be defined as verb + attribute, where the list of verbs is defined separately from the list of attributes.

The second item reduces the amount of text needed to define a device, since all operations on a single attribute (get, set, monitor, etc.) may be defined in a single line as long as they all use the same serviceData.

For this initial specification, the class definition contains 4 parts:

- inheritance specification
- verbs
- attributes
- messages

For each attribute or message there are 3 parts:

- attribute or message text
- service name
- service data

Service data is zero or more (name=value) pairs, where the allowed set of names are defined by the “tags” clause in the service specification. The value may contain a pair of angle brackets “<>” into which the device name will be substituted.

Example: Suppose there are a set of magnets for which it is possible to read and write bdl (integral field), and current (amps). That is, the messages “get bdl”, “get current”, “set bdl”, etc. are valid. Also, each magnet responds to the commands “on” and “off”. Further suppose that the beamline position and length of the magnet are available in a static database (assume read/write).

Here is a complete class definition:

Figure 21: Sample inherited class definition

```
class stdio
    {
    verbs {get, set, monitorOn, monitorOff}
    }

class magnet : stdio
    {
    attributes
        {
        bdl ca {pv=<>.bdl};
        current ca {pv=<>.val};
        zpos os {path=<>/phy/z};
        length os {path=<>/phy/len}
        }
    messages
        {
        on  ca {pv=<>CSR.val, default=1};
        off  ca {pv=<>CSR.val, default=0};
        }
    }
```

Note the semicolon at the end of each line of attributes or messages (except for the last line). Note also that the process variable name has a place holder into which the device name will be substituted. More sophisticated name mangling is envisaged, but not for this release. Syntax is still open to revision.

Instance Definition

Instances of a class are given by following the class name with a list of instance names. Device names may be separated by whitespace characters (including newline) or by commas.

Figure 22: Multiple instances of the magnet class

```
magnet : m1 m2 m3;
```

Combined with the previous definition, the message “on” sent to magnet m2 would select the process variable “m2CSR.val” with “1” as the value to write.

Aliases

Sometimes it is convenient to refer to a device by more than one name. The DDL syntax supports simple aliases, one per line.

Figure 23: Alias device name

```
alias myname m1
```

6. Channel Access Service

EPICS / Channel Access version 3.12 is recommended.

Features:

- Supports synchronous and asynchronous send's.
- Tracks file descriptor registration.
- Fetches data in native type (type conversion done on client and not server).
- Provides a default name service so that if a device is not defined, then an EPICS channel access connection is attempted with the record name set equal to the device name, and the field name set equal to the attribute name.

Restrictions:

- Only supports set/get/monitorOn/monitorOff verbs and the pv and readonly tags. Support for arbitrary messages with default data will be in the next release.
- Discards all exception callbacks from channel access in this version.
- Channel access security is not supported in this release.

Compilation options:

- _CA_SYNC_CONN = perform connections synchronously, waiting up to 4 seconds.
- _CDEV_DEBUG = print verbose messages
- _EPICS_3_12 = if not defined, use 3.11 calls instead.

Introduction to CDEV

Overview of the Control Device Interface

Chip Watson, Jie Chen, Danjin Wu, Walt Akers

Version 1.5 December 9, 1996

TJNAF - Thomas Jefferson National Accelerator Facility

DOCUMENT DATE: Table of ontents generated: December 9, 1996 11:45 am

TRADEMARKS:

SURA/CEBAF:

The Southeastern Universities Research Association (SURA) operates the Continuous Electron Beam Accelerator Facility (CEBAF) for the United States Department of Energy under contract DE-AC05-84ER40150.

DISCLAIMER:

Table of Contents

List of Figures

1. Introduction

What is CDEV

The CDEV (common device) C++ class library is an object-oriented framework that provides a standard interface between an application and one or more underlying control packages or systems. Each underlying package (called a service) is dynamically loaded at run-time, allowing an application to access a new service by name without re-compilation or re-linking. A major feature of CDEV is its ability to hide the implementation details of a particular device from the high level application developer. This allows server implementation choices to be changed without breaking high level client applications. Additionally, because the application is unaware of the underlying service's specific implementation, CDEV is a powerful vehicle for developing portable applications.

The basic CDEV interface can be described through a *device*/*message* paradigm. The *device* is not necessarily a physical object, rather, it is a cdevDevice object that provides a standard interface to CDEV. The application uses this interface to send messages to the device. When the cdevDevice receives this message it will dynamically load the appropriate service (as defined in the CDEV device definition file), and forward the message to it. The service may then perform any action necessary to satisfy the application's request.

The CDEV API is optimized for repetitive operations, in particular for monitoring value changes within a server. Asychronous operations with callbacks are supported, including asychronous error notification. Specifically, CDEV supports most features of channel access (the EPICS communications protocol).

Layering and Abstraction

The CDEV library provides a standard interface to one or more underlying control packages or systems. Currently CDEV provides support for EPICS channel access, tcl scripts, the ACE communications package (a C++ class library over TCP/IP). The ACE package is used to communicate with an on-line accelerator modeling package and with CODA, the CEBAF On-line Data Acquisition system. Access to the EPICS archive data will be added in a later release. All data will be available through a single API (application programming interface).

Information Hiding

The ability to hide the implementation details of a particular device (such as the choice of records in EPICS) from the high level application developer, allows server implementation choices to be changed without breaking all high level client applications. For example, several records may be merged into a new custom record without changing applications which access those records through the CDEV API.

Device Object

The basic idea behind CDEV is that all I/O is performed by sending messages to devices. A device is a named entity in the control system which can respond to a set of messages such as *on* or *off* or *get current*. Further, a device is a virtual entity potentially spanning multiple servers and services such as channel access, an archiver, and a static database.

Each device and message pair is mapped to a unique service (such as channel access) and service dependent data (such as the process variable name in EPICS). This mapping is kept in a name service, and is ultimately initialized from an ASCII file. The designer/maintainer of the server would typically be the person responsible for keeping the mapping current.

Standard Messages

Most devices will implement a number of attributes which may be read or written. In order to facilitate developing generic applications, the following standard messages are recommended for devices, where *attrib* is some attribute of the device:

```
set attrib            => Set the value of the attribute
get attrib            => Get the value of the attribute
monitorOn attrib     => Start monitoring the value of the attribute
monitorOff attrib     => Stop monitoring the value of the attribute
```

Data Object

The message sent to the device may have an associated data object. For example, the message *set current* will have an associated *value* for the new current setting. This value is passed as an additional argument on the send call.

Data returned from the server to the client may contain multiple tagged values, including *value*, *status*, and *timestamp*. Each of these items is extracted from the returned data object by specifying a name or an integer tag. Any data object may contain an arbitrary number of tagged entries, allowing considerable flexibility in defining future clients and servers.

Acknowledgment: This work derives some of its ideas about devices and messages from work done at Argonne by Claude Saunders on a device oriented layer above channel access. Much additional analysis & design work has been done by members of the EPICS collaboration.

2. Basic Operations on Devices and Data

The basic user deals only with the device and data objects. The device is created by name, and I/O consists of sending messages to this device. The device object is the simplest I/O object to use: a single device (magnet, viewer, etc.) may be completely controlled from a single device object, with data being passed by 2 data objects (one for output, another for returned results; dataless commands would not require either). The following sample code reads the magnet current from one magnet, and sets the magnet current for a second magnet to that value:

Figure 1: Sample usage of the send method of a cdevDevice

```
void main ()
    {
    // ************************************************************
    // * Get references to the 2 devices
    // ************************************************************
    cdevDevice& mag1 = cdevDevice::attachRef ("Magnet1");
    cdevDevice& mag2 = cdevDevice::attachRef ("Magnet2");
    cdevData result;

    // ************************************************************
    // * Get the current from the first magnet and use it to set
    // * the current on the second.
    // ************************************************************
    mag1.send("get current", NULL, result);
    mag2.send("set current", result, NULL);
    }
```

The send call, which is synchronous, takes 3 arguments, and returns an integer status:

```
int send(char* msg, cdevData& out, cdevData& result);
```

Like all CDEV calls, a return code of 0 means success (error handling is described in a later section). The first argument to *send* is a character string message. The device name and this string uniquely determine the underlying service to use, and whatever addressing data that service needs to locate the server. The first *send* operation automatically connects to the requested service, initializing any underlying packages as needed.

The second and third arguments are outbound and result data of type cdevData. This is a composite data object which may contain an arbitrary number of data values tagged with an integer tag. The data is self-describing to the extent that cdevData keeps track of the data type, and can automatically convert between data types through C++ function overloading or by explicit direction from the caller. The integer tags are not known to the user at compile time (i.e., no header files required), and can be referred to by character string equivalents. The cdevData class keeps track of all integer tags and their character string equivalents. Character string tags may be converted to integer tags at run time to improve performance. The following code demonstrates how to insert values into and extract values from a cdevData object:

Figure 2: Overview of cdevData operations

```
cdevData d1, d2;
int valtag;
float x, x2;
int status;
...
d1.tagC2I(“value”,&valtag);      // get the integer tag for “value”
...
d1.insert(“value”,37.9);         // insert using a character tag
device.send(“message”,d1,d2);    // send d1, get back d2
d2.get(valtag,&x);               // get “value” from d2 into x
d2.get(“status”,&status);        // extract the “status”
x2 = d2;                         // get “value” into x2 (= overload)
```

The last line above demonstrates using operator overloading to extract the “value” item from the data object; i.e., the character tag “value” is the default tag to use in extracting a data item for the “=” function overload. This form is available for all scalar types supported by cdevData.

Most I/O operations use the “value” tag to transmit a single value (scalar or array). Other commonly used tags are “status” and “time” (a POSIX time struct). These tagged data items are optionally returned by some servers under the control of the I/O context (see below for changing the default context).

In this release, cdevData supports the following data types: *unsigned char, short, unsigned short, int, unsigned int, long, unsigned long, float, double*, *character strings* and *time stamp.* All data types may be either scalar or multi-dimensional arrays.

3. Asychronous Operations

In addition to the synchronous send, there are 2 asynchronous forms for sending messages. The class definition for all three forms is given below.

Figure 3: Three forms of the send command

```
class cdevDevice:
{
public:
    int send (char* msg, cdevData& out, cdevData& result);
    int sendNoBlock (char* msg, cdevData& out, cdevData& result);
    int sendCallback (char* msg, cdevData& out, cdevCallback cb);
};
```

The second form, sendNoBlock, takes the same arguments as send but completes asynchronously. That is, the result will be invalid until some synchronizing action is taken (discussed below). The third form returns the result to a callback function specified as part of the callback argument. For each of these asychronous calls, the message may be buffered by the underlying service, so that upon return from the call there is no guarantee that the message has even been sent yet. Transmission may be forced with a flush call using the cdevSystem object described below. A flush is automatically performed if pend is called.

Callback Object

The callback object is a simple object containing 2 items: a function pointer, and an arbitrary user argument. As an example, the following code fragment demonstrates monitoring the magnet current asynchronously (the third argument to the callback function is discussed later):

Figure 4: Overview of the cdevCallback object and the cdevCallbackFunction

```
// declare the callback func
cdevCallbackFunction gotit;

// create callback object
cdevCallback callbk(gotit, myarg);
...
mag1.sendCallback(“monitorOn current”,NULL,callbk);
...

void gotit(int status, void* userarg, cdevRequestObject& reqobj,
          cdevData& result)
    {
    float f = result;
    printf(“new value is %f\n”,f);
    }
```

The following code defines the callback (the cdevRequestObject is described below):

Figure 5: Structure of the cdevCallback object

```
typedef void (*cdevCallbackFunction)(int status, void* userarg,
                                     cdevRequestObject& reqobj,
                                     cdevData& result);
class cdevCallback
{
public:
   cdevCallback (cdevCallbackFunction func, void* userarg);
   cdevCallbackFunction function;
   void* userarg;
}
```

System Object

Waiting for completion of asychronous operations may be accomplished in one of two ways: (1) using groups, and (2) using the system object. The system object keeps track of all devices, and contains methods for flushing, polling, and pending for asychronous operations.

Figure 6: Usage of the pend method in the cdevSystem object

```
// get default system
cdevSystem& sys = cdevSystem::defaultSystem();
...
mag1.sendNoBlock("on",...);  // async op(s)
mag2.sendNoBlock("on",...);  // async op(s)
...
sys.pend();                  // wait for all I/O to complete
```

The following class interface shows the forms for each synchronization method:

Figure 7: Synchronization methods in the cdevSystem object

```
class cdevSystem
{
public:
   int flush();
   int poll();
   int pend();
}
```

flush: Flush all pending output to the network for those services that perform send buffering.

poll: Flush all pending output, and process all received replies (including dispatching callbacks, if any). Any received monitor callbacks will be delivered as well.

pend: Flush all pending output, and process all replies for the specified number of seconds. If the time argument is omitted, wait until all replies have been received. If a monitor operation was started, this waits for the connection (first callback) only.

Group Object

An alternative mechanism for waiting for asychronous calls is through the use of groups. A group keeps track off all I/O operations started from the time the group is started until the group is ended. Groups may be nested or overlapped, and support the same flush/poll/pend operations as the system object. For example:

Figure 8: Sample usage of the cdevGroup object

```
cdevGroup g1, g2;
...
g1.start();
mag1.sendCallback(...);
mag2.sendCallback(...);
g2.start();
bpm1.sendNoBlock(...);
bpm2.sendNoBlock(...);
g1.end();
mag3.sendCallback(...);
bpm3.sendNoBlock(...);
g2.end();
g1.pend(4.0);
// at this point operations on mag1, mag2, bpm1, & bpm2 are done
// or have timed out (error handling discussed later).
...
// mag3 and bpm3 may not be finished yet, pend on the second
// group to wait for their completion
g2.pend();
```

Note: Allowing nested or overlapped groups allows library calls to start or stop groups without needing to know if another group is already active.

Groups may be operated in one of two modes. Immediate mode (default) causes grouped operations to be immediately executed, and the group is just used for completion synchronization. Deferred mode causes messages to be held back until the group is ended and the operations are flushed. After the operations are complete, he group of operations may be flushed again (without re-posting), allowing the group to function as a list manager/executor.

Request Object

Occasionally, an I/O operation may need to be repeated may times. In this case, it is not efficient to parse the message each time to determine which server to use. It is possible to bypass this parsing by creating a request object. The request object is like the device object, except that the message string is specified at creation time, and the request object is therefore specific to the particular underlying service to which that message must be directed.

In the case of EPICS channel access, the request object opens and maintains the channel to the EPICS process variable. The request object is NOT the same as an EPICS channel, in that the request object binds a device and a message, and the

message implies a direction. If the message is considered to be of the form *verb + attribute* then the EPICS channel essentially binds device and attribute but not verb. In some future release of CDEV, a more channel like object may be included if there is a demand for it. If two cdevRequestObjects reference the same EPICS channel, only a single channel access connection is established.

The following code demonstrates sending a set of data values to a single magnet using a request object. Note that the send call omits the message argument:

Figure 9: Sample usage of the cdevRequestObject object

```
cdevRequestObject& mag1BDL =
    cdevRequestObject::attachRef("mag1","set BDL");
cdevData mydata;
...
for (i=0;i<100;i++)
    {
    mydata = i*10;              // 10 amp steps
    mag1BDL.send(mydata,NULL);  // ignore errors for now
    sleep(1);
    }
```

In most cases, the device object will create the request object to perform a requested I/O operation. All 3 forms of send are supported by the request object, with a calling syntax identical to the device calls without the message argument. The interface to the request object is given below:

Figure 10: Structure of the cdevRequestObject object

```
class cdevRequestObject
{
public:
    char* message();            // returns the message string
    cdevDevice& device();       // returns the device object
    int state();                // connected/disconnected/
    int access();               // read/write/none
    int send(cdevData& out, cdevData& result);
    int sendNoBlock(cdevData& out, cdevData& result);
    int sendCallback(cdevData& out, cdevCallback callback);
}
```

When an asynchronous operation completes, the relevant request object is passed to the user's callback routine. This allows the user to extract the message string and device name from the request object if necessary or desired.

File Descriptors and "select"

An alternative mechanism for dealing with asynchronous operations is to directly test the file descriptors being used. This interface is currently implemented in the cdevSystem class. The pertinent methods have the following form:

Figure 11: File descriptor routines of the cdevSystem object

```
typedef void (*cdevFdChangedCallback)(int fd, int opened,
                void* userarg)
class cdevSystem
{
public:
    ...
    int getFd(int fd[], int &numFD);
    int addFdChangedCallback(cdevFdChangedCallback cbk, void* arg);
}
```

The getFd method will populate a user allocated array of integers (fd) with the file descriptors currently in use by the cdevSystem object. The number of file descriptors that were allocated by the user is specified in the numFD variable. Upon completion, the numFD variable will be set to the number of file descriptors actually copied into the fd array. An error is returned if the buffer is not sufficient to store the complete set of file descriptors.

The addFdChangedCallback method allows the user to specify a function to be called each time a file descriptor is added (opened=1) or closed (opened=0).

The following code illustrates the usage of the UNIX select call:

Figure 12: Using CDEV file descriptors with the UNIX select function

```
int myFD[5];

void mySelectLoop ( cdevSystem & system )
   {
   int fd[20];
   int numFD = 15, nfds;
   fd_set rfds; // Ready file descriptors
   fd_set afds; // Active file descriptors

   // Copy the file descriptors I am already using to the list
   memcpy(fd, myFD, sizeof(myFD));

   // Get the file descriptors in use by the cdevSystem object
   system.getFd(&fd[5], numFD);

   // Add in the 5 previously defined file descriptors
   numFD += 5;

   // Zero the active file descriptors
   FD_ZERO(&afds);

   // Setup the active file descriptors
   for(int i=0; i<numFD; i++) FD_SET(fd[i], &afds);

   // Get the maximum number of file descriptors
   nfds = FD_SETSIZE;

   while(1) {
      // Copy the active descriptors to the ready descriptors
      memcpy((void *)&rfds, (void *)&afds, sizeof(rfds));

      // Use select to detect activity on the file descriptors
      if(select(nfds, &rfds, 0, 0, 0)<0) exit(1);

      // Iterate through the list of file descriptors
      for(i=0; i<numFD, i++)
         {
         if (FD_ISSET(fd[i], &rfds)
            {
            // Respond to active file descriptor here
            }
         }
      }
   }
```

4. Error Handling

Most CDEV routines return an integer status, with 0=success. Errors may also be automatically printed or routed to a user supplied error handler. Control over this behavior is through the system object:

Figure 13: Error handling mechanisms provided by the cdevSystem object

```
typedef void (*cdevErrorHandler)(int severity, char* text,
                                 cdevRequestObject& request);
class cdevSystem
{
public:
    ...
    int autoErrorOn();
    int autoErrorOff();
    cdevErrorHandler setErrorHandler(cdevErrorHandler handler = 0);
    int reportError(int severity, char *name,
                    cdevRequestObject* request, char* format, ...);
    void setThreshold ( int errorThreshold );
}
```

The *severity* parameter indicates the level of the error. Severity levels are informative, warning, error, and severe error. The following values are defined in cdevErrCode.h to describe the severity of an error message:

Figure 14: Enumerated severity codes generated by CDEV

CDEV_SEVERITY_INFO	**Informative message**
CDEV_SEVERITY_WARN	**Warning message –** operation encountered a problem during processing.
CDEV_SEVERITY_ERROR	**Error message –** operation cannot be completed
CDEV_SEVERITY_SEVERE	**Severe/Fatal Error –** cdev cannot continue execution

autoErrorOn Turn on default error handling, which prints error to stdout.

autoErrorOff Turn off default error handling.

setErrorHandler Set a new function to be the system error handler. Return the old function pointer. Omitting the argument resets the handler to the default handler which simply prints error messages to stderr.

reportError Routine which behaves like printf with 3 additional arguments (severity, caller's name, and request object if available).

Note: For simplicity of use, a very compact set of error codes is implemented in CDEV. The following error codes are defined in the header file cdevErrCode.h.

Figure 15: Enumerated error codes generated by CDEV

```
CDEV_WARNING         = -2  Failure of function is non-consequential
CDEV_ERROR           = -1  Errors that are not in any categories
CDEV_SUCCESS         = 0   Success
CDEV_INVALIDOBJ      = 1   Invalid cdev objects
CDEV_INVALIDARG      = 2   Invalid argument passed to cdev calls
CDEV_INVALIDSVC      = 3   Wrong service during dynamic loading
CDEV_INVALIDOP       = 4   Operation is unsupported (collection)
CDEV_NOTCONNECTED    = 5   Not connected to low network service
CDEV_IOFAILED        = 6   Low level network service IO failed
CDEV_CONFLICT        = 7   Conflicts of data types or tags
CDEV_NOTFOUND        = 8   Cannot find user request (cdevData)
CDEV_TIMEOUT         = 9   Time out
CDEV_CONVERT         = 10  cdevData Conversion error
CDEV_OUTOFRANGE      = 11  Value out of range for device attribute
CDEV_NOACCESS        = 12  Insufficient access to perform request
CDEV_ACCESSCHANGED   = 13  Change in access permission of device
CDEV_DISCONNECTED    = 60  Application has lost server connection
CDEV_RECONNECTED     = 61  Application has regained server connection
```

5. Name Resolution

Some type of name resolution system is needed to (1) locate which package (called a service) underneath CDEV will support the specified message for the specified device and (2) provide parameters needed by the service to contact the server and perform the desired operation. For example, the message “get current” sent to a magnet “mag1” might use the service “ca” (channel access), with parameter “mag1cur.val” (record and field). This parameter is referred to as service data, i.e. data used by the service to perform the operation. The user is typically unaware of this data, which should be supplied by the device implementer.

cdevDirectory Object

Name resolution in CDEV is implemented by the *cdevDirectory* device. The cdevDirectory device supports a *query* message allowing the user to search the Device Directory List (DDL) for devices which are (1) members of a user specified class or (2) match a user specified regular expression.

As with any cdevDevice object, the user instantiates a cdevDirectory device by using the *cdevDevice::attachPtr()* or *cdevDevice::attachRef()* method. The following code fragment illustrates the correct method for attaching to a cdevDirectory device.

Figure 16: Obtaining a reference to the cdevDirectory device

```
cdevDevice & device = cdevDevice::attachRef(“cdevDirectory”);
```

Like all other devices, the cdevDirectory response to messages. The following messages may be submitted to this device:

- query: Identify the devices that match the selection criteria.
- queryClass: Identify the DDL class from which a device is instantiated.
- queryAttributes: Identify all attributes supported by a device or a DDL class.
- queryMessages: Identify all messages supported by a device or DDL class.
- queryVerbs: Identify all verbs supported by a device or DDL class.
- service: Identify the service that is used by a device/message pair.
- serviceData: Identify service data specified for a device/message pair.
- update: Add information to the cdevDirectory data structure.
- validate: Verify that a device or DDL class contains certain definitions.

By default, there is a single directory device. To allow for arbitrary expansion CDEV will allow each service to register additional directory devices with the system object. A search list of directories may be specified to the system directory device (planned feature).

Each directory acts as if it has a table of information in the following form:

Figure 17: Virtual form of the data in a cdevDirectory

dev-class	dev-name	message	service	service-data
magnet	mag1	on	ca	{pv=M1CSR.VAL default=1}

This table is loaded from a device definition file which describes the mapping from (device,message) to (service,serviceData), where “service” is one of the dynamically loaded CDEV services, and “serviceData” is whatever information that service needs to send the message to the device. In this example the service data contains an EPICS process variable name and a default value to write to that channel.

It is NOT the purpose of the device definition file to completely specify the datatypes of all data passed between device and user. CDEV does considerable data type conversion as needed, and services do reasonable run-time validation. The goal is to make the file as small as is reasonably possible and still maintain readability.

General:

- leading whitespace is not significant
- any amount of whitespace may separate language elements, including space, tab, and newline, with the exception that at least one whitespace character is needed before the colon separating a class name and its inherited classes.
- keywords are case sensitive

NOTE: This specification is preliminary, and will be expanded to support additional features as time allows.

The interface definition consists of three parts: service definitions, device class definitions, and device instantiation (in that order). Standard definitions may be specified via include files using a cpp syntax:

Figure 18: Syntax of the #include directive

```
#include “filename”
```

Service Definition

The service definition declares a service name, and lists all tags which the service will accept for its “serviceData”. The following 3 lines define a service named “myservice” which supports serviceData that specifies values for any of the tags “abc”, “def”, and “ghi”:

Figure 19: Sample service definition

```
service myservice
    {
    tags {abc, def, ghi}
    }
```

As a more concrete example, the following is the specification for the channel access service:

Figure 20: Sample service definition

```
service ca
    {
    tags {pv, default}
    }
```

The “pv” tag is used to specify the process variable name, and the “default” tag is used to specify a default value for a “set” operation (examples are below).

Device Class Definition

Device classes are used to define a collection of similar devices; that is, devices which will respond to the same set of messages. In order to simplify class definition, the following design choices are made:

- multiple inheritance is supported, and
- messages may be defined as verb + attribute, where the list of verbs is defined separately from the list of attributes.

The second item reduces the amount of text needed to define a device, since all operations on a single attribute (get, set, monitor, etc.) may be defined in a single line as long as they all use the same serviceData.

For this initial specification, the class definition contains 4 parts:

- inheritance specification
- verbs
- attributes
- messages

For each attribute or message there are 3 parts:

- attribute or message text
- service name
- service data

Service data is zero or more (name=value) pairs, where the allowed set of names are defined by the “tags” clause in the service specification. The value may contain a pair of angle brackets “<>” into which the device name will be substituted.

Example: Suppose there are a set of magnets for which it is possible to read and write bdl (integral field), and current (amps). That is, the messages “get bdl”, “get current”, “set bdl”, etc. are valid. Also, each magnet responds to the commands “on” and “off”. Further suppose that the beamline position and length of the magnet are available in a static database (assume read/write).

Here is a complete class definition:

Figure 21: Sample inherited class definition

```
class stdio
    {
    verbs {get, set, monitorOn, monitorOff}
    }

class magnet : stdio
    {
    attributes
        {
        bdl ca {pv=<>.bdl};
        current ca {pv=<>.val};
        zpos os {path=<>/phy/z};
        length os {path=<>/phy/len}
        }
    messages
        {
        on  ca {pv=<>CSR.val, default=1};
        off  ca {pv=<>CSR.val, default=0};
        }
    }
```

Note the semicolon at the end of each line of attributes or messages (except for the last line). Note also that the process variable name has a place holder into which the device name will be substituted. More sophisticated name mangling is envisaged, but not for this release. Syntax is still open to revision.

Instance Definition

Instances of a class are given by following the class name with a list of instance names. Device names may be separated by whitespace characters (including newline) or by commas.

Figure 22: Multiple instances of the magnet class

```
magnet : m1 m2 m3;
```

Combined with the previous definition, the message “on” sent to magnet m2 would select the process variable “m2CSR.val” with “1” as the value to write.

Aliases

Sometimes it is convenient to refer to a device by more than one name. The DDL syntax supports simple aliases, one per line.

Figure 23: Alias device name

```
alias myname m1
```

6. Channel Access Service

EPICS / Channel Access version 3.12 is recommended.

Features:

- Supports synchronous and asynchronous send's.
- Tracks file descriptor registration.
- Fetches data in native type (type conversion done on client and not server).
- Provides a default name service so that if a device is not defined, then an EPICS channel access connection is attempted with the record name set equal to the device name, and the field name set equal to the attribute name.

Restrictions:

- Only supports set/get/monitorOn/monitorOff verbs and the pv and readonly tags. Support for arbitrary messages with default data will be in the next release.
- Discards all exception callbacks from channel access in this version.
- Channel access security is not supported in this release.

Compilation options:

- _CA_SYNC_CONN = perform connections synchronously, waiting up to 4 seconds.
- _CDEV_DEBUG = print verbose messages
- _EPICS_3_12 = if not defined, use 3.11 calls instead.

Introduction to CDEV

Overview of the Control Device Interface

Chip Watson, Jie Chen, Danjin Wu, Walt Akers

Version 1.5 December 9, 1996

TJNAF - Thomas Jefferson National Accelerator Facility

DOCUMENT DATE: Table of ontents generated: December 9, 1996 11:45 am

TRADEMARKS:

SURA/CEBAF:

The Southeastern Universities Research Association (SURA) operates the Continuous Electron Beam Accelerator Facility (CEBAF) for the United States Department of Energy under contract DE-AC05-84ER40150.

DISCLAIMER:

Table of Contents

List of Figures

1. Introduction

What is CDEV

The CDEV (common device) C++ class library is an object-oriented framework that provides a standard interface between an application and one or more underlying control packages or systems. Each underlying package (called a service) is dynamically loaded at run-time, allowing an application to access a new service by name without re-compilation or re-linking. A major feature of CDEV is its ability to hide the implementation details of a particular device from the high level application developer. This allows server implementation choices to be changed without breaking high level client applications. Additionally, because the application is unaware of the underlying service's specific implementation, CDEV is a powerful vehicle for developing portable applications.

The basic CDEV interface can be described through a *device*/*message* paradigm. The *device* is not necessarily a physical object, rather, it is a cdevDevice object that provides a standard interface to CDEV. The application uses this interface to send messages to the device. When the cdevDevice receives this message it will dynamically load the appropriate service (as defined in the CDEV device definition file), and forward the message to it. The service may then perform any action necessary to satisfy the application's request.

The CDEV API is optimized for repetitive operations, in particular for monitoring value changes within a server. Asychronous operations with callbacks are supported, including asychronous error notification. Specifically, CDEV supports most features of channel access (the EPICS communications protocol).

Layering and Abstraction

The CDEV library provides a standard interface to one or more underlying control packages or systems. Currently CDEV provides support for EPICS channel access, tcl scripts, the ACE communications package (a C++ class library over TCP/IP). The ACE package is used to communicate with an on-line accelerator modeling package and with CODA, the CEBAF On-line Data Acquisition system. Access to the EPICS archive data will be added in a later release. All data will be available through a single API (application programming interface).

Information Hiding

The ability to hide the implementation details of a particular device (such as the choice of records in EPICS) from the high level application developer, allows server implementation choices to be changed without breaking all high level client applications. For example, several records may be merged into a new custom record without changing applications which access those records through the CDEV API.

Device Object

The basic idea behind CDEV is that all I/O is performed by sending messages to devices. A device is a named entity in the control system which can respond to a set of messages such as *on* or *off* or *get current*. Further, a device is a virtual entity potentially spanning multiple servers and services such as channel access, an archiver, and a static database.

Each device and message pair is mapped to a unique service (such as channel access) and service dependent data (such as the process variable name in EPICS). This mapping is kept in a name service, and is ultimately initialized from an ASCII file. The designer/maintainer of the server would typically be the person responsible for keeping the mapping current.

Standard Messages

Most devices will implement a number of attributes which may be read or written. In order to facilitate developing generic applications, the following standard messages are recommended for devices, where *attrib* is some attribute of the device:

```
set attrib            => Set the value of the attribute
get attrib            => Get the value of the attribute
monitorOn attrib     => Start monitoring the value of the attribute
monitorOff attrib     => Stop monitoring the value of the attribute
```

Data Object

The message sent to the device may have an associated data object. For example, the message *set current* will have an associated *value* for the new current setting. This value is passed as an additional argument on the send call.

Data returned from the server to the client may contain multiple tagged values, including *value*, *status*, and *timestamp*. Each of these items is extracted from the returned data object by specifying a name or an integer tag. Any data object may contain an arbitrary number of tagged entries, allowing considerable flexibility in defining future clients and servers.

Acknowledgment: This work derives some of its ideas about devices and messages from work done at Argonne by Claude Saunders on a device oriented layer above channel access. Much additional analysis & design work has been done by members of the EPICS collaboration.

2. Basic Operations on Devices and Data

The basic user deals only with the device and data objects. The device is created by name, and I/O consists of sending messages to this device. The device object is the simplest I/O object to use: a single device (magnet, viewer, etc.) may be completely controlled from a single device object, with data being passed by 2 data objects (one for output, another for returned results; dataless commands would not require either). The following sample code reads the magnet current from one magnet, and sets the magnet current for a second magnet to that value:

Figure 1: Sample usage of the send method of a cdevDevice

```
void main ()
    {
    // ************************************************************
    // * Get references to the 2 devices
    // ************************************************************
    cdevDevice& mag1 = cdevDevice::attachRef ("Magnet1");
    cdevDevice& mag2 = cdevDevice::attachRef ("Magnet2");
    cdevData result;

    // ************************************************************
    // * Get the current from the first magnet and use it to set
    // * the current on the second.
    // ************************************************************
    mag1.send("get current", NULL, result);
    mag2.send("set current", result, NULL);
    }
```

The send call, which is synchronous, takes 3 arguments, and returns an integer status:

```
int send(char* msg, cdevData& out, cdevData& result);
```

Like all CDEV calls, a return code of 0 means success (error handling is described in a later section). The first argument to *send* is a character string message. The device name and this string uniquely determine the underlying service to use, and whatever addressing data that service needs to locate the server. The first *send* operation automatically connects to the requested service, initializing any underlying packages as needed.

The second and third arguments are outbound and result data of type cdevData. This is a composite data object which may contain an arbitrary number of data values tagged with an integer tag. The data is self-describing to the extent that cdevData keeps track of the data type, and can automatically convert between data types through C++ function overloading or by explicit direction from the caller. The integer tags are not known to the user at compile time (i.e., no header files required), and can be referred to by character string equivalents. The cdevData class keeps track of all integer tags and their character string equivalents. Character string tags may be converted to integer tags at run time to improve performance. The following code demonstrates how to insert values into and extract values from a cdevData object:

Figure 2: Overview of cdevData operations

```
cdevData d1, d2;
int valtag;
float x, x2;
int status;
...
d1.tagC2I("value",&valtag);      // get the integer tag for "value"
...
d1.insert("value",37.9);         // insert using a character tag
device.send("message",d1,d2);    // send d1, get back d2
d2.get(valtag,&x);               // get "value" from d2 into x
d2.get("status",&status);        // extract the "status"
x2 = d2;                         // get "value" into x2 (= overload)
```

The last line above demonstrates using operator overloading to extract the “value” item from the data object; i.e., the character tag “value” is the default tag to use in extracting a data item for the “=” function overload. This form is available for all scalar types supported by cdevData.

Most I/O operations use the “value” tag to transmit a single value (scalar or array). Other commonly used tags are “status” and “time” (a POSIX time struct). These tagged data items are optionally returned by some servers under the control of the I/O context (see below for changing the default context).

In this release, cdevData supports the following data types: *unsigned char, short, unsigned short, int, unsigned int, long, unsigned long, float, double*, *character strings* and *time stamp.* All data types may be either scalar or multi-dimensional arrays.

3. Asychronous Operations

In addition to the synchronous send, there are 2 asynchronous forms for sending messages. The class definition for all three forms is given below.

Figure 3: Three forms of the send command

```
class cdevDevice:
{
public:
    int send (char* msg, cdevData& out, cdevData& result);
    int sendNoBlock (char* msg, cdevData& out, cdevData& result);
    int sendCallback (char* msg, cdevData& out, cdevCallback cb);
};
```

The second form, sendNoBlock, takes the same arguments as send but completes asynchronously. That is, the result will be invalid until some synchronizing action is taken (discussed below). The third form returns the result to a callback function specified as part of the callback argument. For each of these asychronous calls, the message may be buffered by the underlying service, so that upon return from the call there is no guarantee that the message has even been sent yet. Transmission may be forced with a flush call using the cdevSystem object described below. A flush is automatically performed if pend is called.

Callback Object

The callback object is a simple object containing 2 items: a function pointer, and an arbitrary user argument. As an example, the following code fragment demonstrates monitoring the magnet current asynchronously (the third argument to the callback function is discussed later):

Figure 4: Overview of the cdevCallback object and the cdevCallbackFunction

```
// declare the callback func
cdevCallbackFunction gotit;

// create callback object
cdevCallback callbk(gotit, myarg);
...
mag1.sendCallback("monitorOn current",NULL,callbk);
...

void gotit(int status, void* userarg, cdevRequestObject& reqobj,
          cdevData& result)
    {
    float f = result;
    printf("new value is %f\n",f);
    }
```

The following code defines the callback (the cdevRequestObject is described below):

Figure 5: Structure of the cdevCallback object

```
typedef void (*cdevCallbackFunction)(int status, void* userarg,
                                     cdevRequestObject& reqobj,
                                     cdevData& result);
class cdevCallback
{
public:
   cdevCallback (cdevCallbackFunction func, void* userarg);
   cdevCallbackFunction function;
   void* userarg;
}
```

System Object

Waiting for completion of asychronous operations may be accomplished in one of two ways: (1) using groups, and (2) using the system object. The system object keeps track of all devices, and contains methods for flushing, polling, and pending for asychronous operations.

Figure 6: Usage of the pend method in the cdevSystem object

```
// get default system
cdevSystem& sys = cdevSystem::defaultSystem();
...
mag1.sendNoBlock("on",...);  // async op(s)
mag2.sendNoBlock("on",...);  // async op(s)
...
sys.pend();                  // wait for all I/O to complete
```

The following class interface shows the forms for each synchronization method:

Figure 7: Synchronization methods in the cdevSystem object

```
class cdevSystem
{
public:
   int flush();
   int poll();
   int pend();
}
```

flush: Flush all pending output to the network for those services that perform send buffering.

poll: Flush all pending output, and process all received replies (including dispatching callbacks, if any). Any received monitor callbacks will be delivered as well.

pend: Flush all pending output, and process all replies for the specified number of seconds. If the time argument is omitted, wait until all replies have been received. If a monitor operation was started, this waits for the connection (first callback) only.

Group Object

An alternative mechanism for waiting for asychronous calls is through the use of groups. A group keeps track off all I/O operations started from the time the group is started until the group is ended. Groups may be nested or overlapped, and support the same flush/poll/pend operations as the system object. For example:

Figure 8: Sample usage of the cdevGroup object

```
cdevGroup g1, g2;
...
g1.start();
mag1.sendCallback(...);
mag2.sendCallback(...);
g2.start();
bpm1.sendNoBlock(...);
bpm2.sendNoBlock(...);
g1.end();
mag3.sendCallback(...);
bpm3.sendNoBlock(...);
g2.end();
g1.pend(4.0);
// at this point operations on mag1, mag2, bpm1, & bpm2 are done
// or have timed out (error handling discussed later).
...
// mag3 and bpm3 may not be finished yet, pend on the second
// group to wait for their completion
g2.pend();
```

Note: Allowing nested or overlapped groups allows library calls to start or stop groups without needing to know if another group is already active.

Groups may be operated in one of two modes. Immediate mode (default) causes grouped operations to be immediately executed, and the group is just used for completion synchronization. Deferred mode causes messages to be held back until the group is ended and the operations are flushed. After the operations are complete, he group of operations may be flushed again (without re-posting), allowing the group to function as a list manager/executor.

Request Object

Occasionally, an I/O operation may need to be repeated may times. In this case, it is not efficient to parse the message each time to determine which server to use. It is possible to bypass this parsing by creating a request object. The request object is like the device object, except that the message string is specified at creation time, and the request object is therefore specific to the particular underlying service to which that message must be directed.

In the case of EPICS channel access, the request object opens and maintains the channel to the EPICS process variable. The request object is NOT the same as an EPICS channel, in that the request object binds a device and a message, and the

message implies a direction. If the message is considered to be of the form *verb + attribute* then the EPICS channel essentially binds device and attribute but not verb. In some future release of CDEV, a more channel like object may be included if there is a demand for it. If two cdevRequestObjects reference the same EPICS channel, only a single channel access connection is established.

The following code demonstrates sending a set of data values to a single magnet using a request object. Note that the send call omits the message argument:

Figure 9: Sample usage of the cdevRequestObject object

```
cdevRequestObject& mag1BDL =
    cdevRequestObject::attachRef("mag1","set BDL");
cdevData mydata;
...
for (i=0;i<100;i++)
    {
    mydata = i*10;              // 10 amp steps
    mag1BDL.send(mydata,NULL);  // ignore errors for now
    sleep(1);
    }
```

In most cases, the device object will create the request object to perform a requested I/O operation. All 3 forms of send are supported by the request object, with a calling syntax identical to the device calls without the message argument. The interface to the request object is given below:

Figure 10: Structure of the cdevRequestObject object

```
class cdevRequestObject
{
public:
    char* message();            // returns the message string
    cdevDevice& device();       // returns the device object
    int state();                // connected/disconnected/
    int access();               // read/write/none
    int send(cdevData& out, cdevData& result);
    int sendNoBlock(cdevData& out, cdevData& result);
    int sendCallback(cdevData& out, cdevCallback callback);
}
```

When an asynchronous operation completes, the relevant request object is passed to the user's callback routine. This allows the user to extract the message string and device name from the request object if necessary or desired.

File Descriptors and "select"

An alternative mechanism for dealing with asynchronous operations is to directly test the file descriptors being used. This interface is currently implemented in the cdevSystem class. The pertinent methods have the following form:

Figure 11: File descriptor routines of the cdevSystem object

```
typedef void (*cdevFdChangedCallback)(int fd, int opened,
                void* userarg)
class cdevSystem
{
public:
    ...
    int getFd(int fd[], int &numFD);
    int addFdChangedCallback(cdevFdChangedCallback cbk, void* arg);
}
```

The getFd method will populate a user allocated array of integers (fd) with the file descriptors currently in use by the cdevSystem object. The number of file descriptors that were allocated by the user is specified in the numFD variable. Upon completion, the numFD variable will be set to the number of file descriptors actually copied into the fd array. An error is returned if the buffer is not sufficient to store the complete set of file descriptors.

The addFdChangedCallback method allows the user to specify a function to be called each time a file descriptor is added (opened=1) or closed (opened=0).

The following code illustrates the usage of the UNIX select call:

Figure 12: Using CDEV file descriptors with the UNIX select function

```
int myFD[5];

void mySelectLoop ( cdevSystem & system )
    {
    int fd[20];
    int numFD = 15, nfds;
    fd_set rfds; // Ready file descriptors
    fd_set afds; // Active file descriptors

    // Copy the file descriptors I am already using to the list
    memcpy(fd, myFD, sizeof(myFD));

    // Get the file descriptors in use by the cdevSystem object
    system.getFd(&fd[5], numFD);

    // Add in the 5 previously defined file descriptors
    numFD += 5;

    // Zero the active file descriptors
    FD_ZERO(&afds);

    // Setup the active file descriptors
    for(int i=0; i<numFD; i++) FD_SET(fd[i], &afds);

    // Get the maximum number of file descriptors
    nfds = FD_SETSIZE;

    while(1) {
        // Copy the active descriptors to the ready descriptors
        memcpy((void *)&rfds, (void *)&afds, sizeof(rfds));

        // Use select to detect activity on the file descriptors
        if(select(nfds, &rfds, 0, 0, 0)<0) exit(1);

        // Iterate through the list of file descriptors
        for(i=0; i<numFD, i++)
            {
            if (FD_ISSET(fd[i], &rfds)
                {
                // Respond to active file descriptor here
                }
            }
        }
    }
```

4. Error Handling

Most CDEV routines return an integer status, with 0=success. Errors may also be automatically printed or routed to a user supplied error handler. Control over this behavior is through the system object:

Figure 13: Error handling mechanisms provided by the cdevSystem object

```
typedef void (*cdevErrorHandler)(int severity, char* text,
                                  cdevRequestObject& request);
class cdevSystem
{
public:
    ...
    int autoErrorOn();
    int autoErrorOff();
    cdevErrorHandler setErrorHandler(cdevErrorHandler handler = 0);
    int reportError(int severity, char *name,
                    cdevRequestObject* request, char* format, ...);
    void setThreshold ( int errorThreshold );
}
```

The *severity* parameter indicates the level of the error. Severity levels are informative, warning, error, and severe error. The following values are defined in cdevErrCode.h to describe the severity of an error message:

Figure 14: Enumerated severity codes generated by CDEV

CDEV_SEVERITY_INFO	**Informative message**
CDEV_SEVERITY_WARN	**Warning message** – operation encountered a problem during processing.
CDEV_SEVERITY_ERROR	**Error message** – operation cannot be completed
CDEV_SEVERITY_SEVERE	**Severe/Fatal Error** – cdev cannot continue execution

autoErrorOn Turn on default error handling, which prints error to stdout.

autoErrorOff Turn off default error handling.

setErrorHandler Set a new function to be the system error handler. Return the old function pointer. Omitting the argument resets the handler to the default handler which simply prints error messages to stderr.

reportError Routine which behaves like printf with 3 additional arguments (severity, caller’s name, and request object if available).

Note: For simplicity of use, a very compact set of error codes is implemented in CDEV. The following error codes are defined in the header file cdevErrCode.h.

Figure 15: Enumerated error codes generated by CDEV

```
CDEV_WARNING          = -2  Failure of function is non-consequential
CDEV_ERROR            = -1  Errors that are not in any categories
CDEV_SUCCESS          = 0   Success
CDEV_INVALIDOBJ       = 1   Invalid cdev objects
CDEV_INVALIDARG       = 2   Invalid argument passed to cdev calls
CDEV_INVALIDSVC       = 3   Wrong service during dynamic loading
CDEV_INVALIDOP        = 4   Operation is unsupported (collection)
CDEV_NOTCONNECTED     = 5   Not connected to low network service
CDEV_IOFAILED         = 6   Low level network service IO failed
CDEV_CONFLICT         = 7   Conflicts of data types or tags
CDEV_NOTFOUND         = 8   Cannot find user request (cdevData)
CDEV_TIMEOUT          = 9   Time out
CDEV_CONVERT          = 10  cdevData Conversion error
CDEV_OUTOFRANGE       = 11  Value out of range for device attribute
CDEV_NOACCESS         = 12  Insufficient access to perform request
CDEV_ACCESSCHANGED    = 13  Change in access permission of device
CDEV_DISCONNECTED     = 60  Application has lost server connection
CDEV_RECONNECTED      = 61  Application has regained server connection
```

5. Name Resolution

Some type of name resolution system is needed to (1) locate which package (called a service) underneath CDEV will support the specified message for the specified device and (2) provide parameters needed by the service to contact the server and perform the desired operation. For example, the message “get current” sent to a magnet “mag1” might use the service “ca” (channel access), with parameter “mag1cur.val” (record and field). This parameter is referred to as service data, i.e. data used by the service to perform the operation. The user is typically unaware of this data, which should be supplied by the device implementer.

cdevDirectory Object

Name resolution in CDEV is implemented by the *cdevDirectory* device. The cdevDirectory device supports a *query* message allowing the user to search the Device Directory List (DDL) for devices which are (1) members of a user specified class or (2) match a user specified regular expression.

As with any cdevDevice object, the user instantiates a cdevDirectory device by using the *cdevDevice::attachPtr()* or *cdevDevice::attachRef()* method. The following code fragment illustrates the correct method for attaching to a cdevDirectory device.

Figure 16: Obtaining a reference to the cdevDirectory device

```
cdevDevice & device = cdevDevice::attachRef(“cdevDirectory”);
```

Like all other devices, the cdevDirectory response to messages. The following messages may be submitted to this device:

- query: Identify the devices that match the selection criteria.
- queryClass: Identify the DDL class from which a device is instantiated.
- queryAttributes: Identify all attributes supported by a device or a DDL class.
- queryMessages: Identify all messages supported by a device or DDL class.
- queryVerbs: Identify all verbs supported by a device or DDL class.
- service: Identify the service that is used by a device/message pair.
- serviceData: Identify service data specified for a device/message pair.
- update: Add information to the cdevDirectory data structure.
- validate: Verify that a device or DDL class contains certain definitions.

By default, there is a single directory device. To allow for arbitrary expansion CDEV will allow each service to register additional directory devices with the system object. A search list of directories may be specified to the system directory device (planned feature).

Each directory acts as if it has a table of information in the following form:

Figure 17: Virtual form of the data in a cdevDirectory

dev-class	dev-name	message	service	service-data
magnet	mag1	on	ca	{pv=M1CSR.VAL default=1}

This table is loaded from a device definition file which describes the mapping from (device,message) to (service,serviceData), where “service” is one of the dynamically loaded CDEV services, and “serviceData” is whatever information that service needs to send the message to the device. In this example the service data contains an EPICS process variable name and a default value to write to that channel.

It is NOT the purpose of the device definition file to completely specify the datatypes of all data passed between device and user. CDEV does considerable data type conversion as needed, and services do reasonable run-time validation. The goal is to make the file as small as is reasonably possible and still maintain readability.

General:

- leading whitespace is not significant
- any amount of whitespace may separate language elements, including space, tab, and newline, with the exception that at least one whitespace character is needed before the colon separating a class name and its inherited classes.
- keywords are case sensitive

NOTE: This specification is preliminary, and will be expanded to support additional features as time allows.

The interface definition consists of three parts: service definitions, device class definitions, and device instantiation (in that order). Standard definitions may be specified via include files using a cpp syntax:

Figure 18: Syntax of the #include directive

```
#include “filename”
```

Service Definition

The service definition declares a service name, and lists all tags which the service will accept for its “serviceData”. The following 3 lines define a service named “myservice” which supports serviceData that specifies values for any of the tags “abc”, “def”, and “ghi”:

Figure 19: Sample service definition

```
service myservice
    {
    tags {abc, def, ghi}
    }
```

As a more concrete example, the following is the specification for the channel access service:

Figure 20: Sample service definition

```
service ca
    {
    tags {pv, default}
    }
```

The “pv” tag is used to specify the process variable name, and the “default” tag is used to specify a default value for a “set” operation (examples are below).

Device Class Definition

Device classes are used to define a collection of similar devices; that is, devices which will respond to the same set of messages. In order to simplify class definition, the following design choices are made:

- multiple inheritance is supported, and
- messages may be defined as verb + attribute, where the list of verbs is defined separately from the list of attributes.

The second item reduces the amount of text needed to define a device, since all operations on a single attribute (get, set, monitor, etc.) may be defined in a single line as long as they all use the same serviceData.

For this initial specification, the class definition contains 4 parts:

- inheritance specification
- verbs
- attributes
- messages

For each attribute or message there are 3 parts:

- attribute or message text
- service name
- service data

Service data is zero or more (name=value) pairs, where the allowed set of names are defined by the “tags” clause in the service specification. The value may contain a pair of angle brackets “<>” into which the device name will be substituted.

Example: Suppose there are a set of magnets for which it is possible to read and write bdl (integral field), and current (amps). That is, the messages “get bdl”, “get current”, “set bdl”, etc. are valid. Also, each magnet responds to the commands “on” and “off”. Further suppose that the beamline position and length of the magnet are available in a static database (assume read/write).

Here is a complete class definition:

Figure 21: Sample inherited class definition

```
class stdio
    {
    verbs {get, set, monitorOn, monitorOff}
    }

class magnet : stdio
    {
    attributes
        {
        bdl ca {pv=<>.bdl};
        current ca {pv=<>.val};
        zpos os {path=<>/phy/z};
        length os {path=<>/phy/len}
        }
    messages
        {
        on  ca {pv=<>CSR.val, default=1};
        off  ca {pv=<>CSR.val, default=0};
        }
    }
```

Note the semicolon at the end of each line of attributes or messages (except for the last line). Note also that the process variable name has a place holder into which the device name will be substituted. More sophisticated name mangling is envisaged, but not for this release. Syntax is still open to revision.

Instance Definition

Instances of a class are given by following the class name with a list of instance names. Device names may be separated by whitespace characters (including newline) or by commas.

Figure 22: Multiple instances of the magnet class

```
magnet : m1 m2 m3;
```

Combined with the previous definition, the message “on” sent to magnet m2 would select the process variable “m2CSR.val” with “1” as the value to write.

Aliases

Sometimes it is convenient to refer to a device by more than one name. The DDL syntax supports simple aliases, one per line.

Figure 23: Alias device name

```
alias myname m1
```

6. Channel Access Service

EPICS / Channel Access version 3.12 is recommended.

Features:

- Supports synchronous and asynchronous send's.
- Tracks file descriptor registration.
- Fetches data in native type (type conversion done on client and not server).
- Provides a default name service so that if a device is not defined, then an EPICS channel access connection is attempted with the record name set equal to the device name, and the field name set equal to the attribute name.

Restrictions:

- Only supports set/get/monitorOn/monitorOff verbs and the pv and readonly tags. Support for arbitrary messages with default data will be in the next release.
- Discards all exception callbacks from channel access in this version.
- Channel access security is not supported in this release.

Compilation options:

- _CA_SYNC_CONN = perform connections synchronously, waiting up to 4 seconds.
- _CDEV_DEBUG = print verbose messages
- _EPICS_3_12 = if not defined, use 3.11 calls instead.

Introduction to CDEV

Overview of the Control Device Interface

Chip Watson, Jie Chen, Danjin Wu, Walt Akers

Version 1.5 December 9, 1996

TJNAF - Thomas Jefferson National Accelerator Facility

DOCUMENT DATE: Table of ontents generated: December 9, 1996 11:45 am

TRADEMARKS:

SURA/CEBAF:

The Southeastern Universities Research Association (SURA) operates the Continuous Electron Beam Accelerator Facility (CEBAF) for the United States Department of Energy under contract DE-AC05-84ER40150.

DISCLAIMER:

Table of Contents

List of Figures

1. Introduction

What is CDEV

The CDEV (common device) C++ class library is an object-oriented framework that provides a standard interface between an application and one or more underlying control packages or systems. Each underlying package (called a service) is dynamically loaded at run-time, allowing an application to access a new service by name without re-compilation or re-linking. A major feature of CDEV is its ability to hide the implementation details of a particular device from the high level application developer. This allows server implementation choices to be changed without breaking high level client applications. Additionally, because the application is unaware of the underlying service's specific implementation, CDEV is a powerful vehicle for developing portable applications.

The basic CDEV interface can be described through a *device*/*message* paradigm. The *device* is not necessarily a physical object, rather, it is a cdevDevice object that provides a standard interface to CDEV. The application uses this interface to send messages to the device. When the cdevDevice receives this message it will dynamically load the appropriate service (as defined in the CDEV device definition file), and forward the message to it. The service may then perform any action necessary to satisfy the application's request.

The CDEV API is optimized for repetitive operations, in particular for monitoring value changes within a server. Asychronous operations with callbacks are supported, including asychronous error notification. Specifically, CDEV supports most features of channel access (the EPICS communications protocol).

Layering and Abstraction

The CDEV library provides a standard interface to one or more underlying control packages or systems. Currently CDEV provides support for EPICS channel access, tcl scripts, the ACE communications package (a C++ class library over TCP/IP). The ACE package is used to communicate with an on-line accelerator modeling package and with CODA, the CEBAF On-line Data Acquisition system. Access to the EPICS archive data will be added in a later release. All data will be available through a single API (application programming interface).

Information Hiding

The ability to hide the implementation details of a particular device (such as the choice of records in EPICS) from the high level application developer, allows server implementation choices to be changed without breaking all high level client applications. For example, several records may be merged into a new custom record without changing applications which access those records through the CDEV API.

Device Object

The basic idea behind CDEV is that all I/O is performed by sending messages to devices. A device is a named entity in the control system which can respond to a set of messages such as *on* or *off* or *get current*. Further, a device is a virtual entity potentially spanning multiple servers and services such as channel access, an archiver, and a static database.

Each device and message pair is mapped to a unique service (such as channel access) and service dependent data (such as the process variable name in EPICS). This mapping is kept in a name service, and is ultimately initialized from an ASCII file. The designer/maintainer of the server would typically be the person responsible for keeping the mapping current.

Standard Messages

Most devices will implement a number of attributes which may be read or written. In order to facilitate developing generic applications, the following standard messages are recommended for devices, where *attrib* is some attribute of the device:

```
set attrib            => Set the value of the attribute
get attrib            => Get the value of the attribute
monitorOn attrib      => Start monitoring the value of the attribute
monitorOff attrib     => Stop monitoring the value of the attribute
```

Data Object

The message sent to the device may have an associated data object. For example, the message *set current* will have an associated *value* for the new current setting. This value is passed as an additional argument on the send call.

Data returned from the server to the client may contain multiple tagged values, including *value*, *status*, and *timestamp*. Each of these items is extracted from the returned data object by specifying a name or an integer tag. Any data object may contain an arbitrary number of tagged entries, allowing considerable flexibility in defining future clients and servers.

Acknowledgment: This work derives some of its ideas about devices and messages from work done at Argonne by Claude Saunders on a device oriented layer above channel access. Much additional analysis & design work has been done by members of the EPICS collaboration.

2. Basic Operations on Devices and Data

The basic user deals only with the device and data objects. The device is created by name, and I/O consists of sending messages to this device. The device object is the simplest I/O object to use: a single device (magnet, viewer, etc.) may be completely controlled from a single device object, with data being passed by 2 data objects (one for output, another for returned results; dataless commands would not require either). The following sample code reads the magnet current from one magnet, and sets the magnet current for a second magnet to that value:

Figure 1: Sample usage of the send method of a cdevDevice

```
void main ()
    {
    // ************************************************************
    // * Get references to the 2 devices
    // ************************************************************
    cdevDevice& mag1 = cdevDevice::attachRef ("Magnet1");
    cdevDevice& mag2 = cdevDevice::attachRef ("Magnet2");
    cdevData result;

    // ************************************************************
    // * Get the current from the first magnet and use it to set
    // * the current on the second.
    // ************************************************************
    mag1.send("get current", NULL, result);
    mag2.send("set current", result, NULL);
    }
```

The send call, which is synchronous, takes 3 arguments, and returns an integer status:

```
int send(char* msg, cdevData& out, cdevData& result);
```

Like all CDEV calls, a return code of 0 means success (error handling is described in a later section). The first argument to *send* is a character string message. The device name and this string uniquely determine the underlying service to use, and whatever addressing data that service needs to locate the server. The first *send* operation automatically connects to the requested service, initializing any underlying packages as needed.

The second and third arguments are outbound and result data of type cdevData. This is a composite data object which may contain an arbitrary number of data values tagged with an integer tag. The data is self-describing to the extent that cdevData keeps track of the data type, and can automatically convert between data types through C++ function overloading or by explicit direction from the caller. The integer tags are not known to the user at compile time (i.e., no header files required), and can be referred to by character string equivalents. The cdevData class keeps track of all integer tags and their character string equivalents. Character string tags may be converted to integer tags at run time to improve performance. The following code demonstrates how to insert values into and extract values from a cdevData object:

Figure 2: Overview of cdevData operations

```
cdevData d1, d2;
int valtag;
float x, x2;
int status;
...
d1.tagC2I("value",&valtag);      // get the integer tag for "value"
...
d1.insert("value",37.9);         // insert using a character tag
device.send("message",d1,d2);    // send d1, get back d2
d2.get(valtag,&x);               // get "value" from d2 into x
d2.get("status",&status);        // extract the "status"
x2 = d2;                         // get "value" into x2 (= overload)
```

The last line above demonstrates using operator overloading to extract the "value" item from the data object; i.e., the character tag "value" is the default tag to use in extracting a data item for the "=" function overload. This form is available for all scalar types supported by cdevData.

Most I/O operations use the "value" tag to transmit a single value (scalar or array). Other commonly used tags are "status" and "time" (a POSIX time struct). These tagged data items are optionally returned by some servers under the control of the I/O context (see below for changing the default context).

In this release, cdevData supports the following data types: *unsigned char, short, unsigned short, int, unsigned int, long, unsigned long, float, double*, *character strings* and *time stamp.* All data types may be either scalar or multi-dimensional arrays.

3. Asychronous Operations

In addition to the synchronous send, there are 2 asynchronous forms for sending messages. The class definition for all three forms is given below.

Figure 3: Three forms of the send command

```
class cdevDevice:
{
public:
    int send (char* msg, cdevData& out, cdevData& result);
    int sendNoBlock (char* msg, cdevData& out, cdevData& result);
    int sendCallback (char* msg, cdevData& out, cdevCallback cb);
};
```

The second form, sendNoBlock, takes the same arguments as send but completes asynchronously. That is, the result will be invalid until some synchronizing action is taken (discussed below). The third form returns the result to a callback function specified as part of the callback argument. For each of these asychronous calls, the message may be buffered by the underlying service, so that upon return from the call there is no guarantee that the message has even been sent yet. Transmission may be forced with a flush call using the cdevSystem object described below. A flush is automatically performed if pend is called.

Callback Object

The callback object is a simple object containing 2 items: a function pointer, and an arbitrary user argument. As an example, the following code fragment demonstrates monitoring the magnet current asynchronously (the third argument to the callback function is discussed later):

Figure 4: Overview of the cdevCallback object and the cdevCallbackFunction

```
// declare the callback func
cdevCallbackFunction gotit;

// create callback object
cdevCallback callbk(gotit, myarg);
...
mag1.sendCallback(“monitorOn current”,NULL,callbk);
...

void gotit(int status, void* userarg, cdevRequestObject& reqobj,
          cdevData& result)
    {
    float f = result;
    printf(“new value is %f\n”,f);
    }
```

The following code defines the callback (the cdevRequestObject is described below):

Figure 5: Structure of the cdevCallback object

```
typedef void (*cdevCallbackFunction)(int status, void* userarg,
                                     cdevRequestObject& reqobj,
                                     cdevData& result);
class cdevCallback
{
public:
   cdevCallback (cdevCallbackFunction func, void* userarg);
   cdevCallbackFunction function;
   void* userarg;
}
```

System Object

Waiting for completion of asychronous operations may be accomplished in one of two ways: (1) using groups, and (2) using the system object. The system object keeps track of all devices, and contains methods for flushing, polling, and pending for asychronous operations.

Figure 6: Usage of the pend method in the cdevSystem object

```
// get default system
cdevSystem& sys = cdevSystem::defaultSystem();
...
mag1.sendNoBlock("on",...);  // async op(s)
mag2.sendNoBlock("on",...);  // async op(s)
...
sys.pend();                  // wait for all I/O to complete
```

The following class interface shows the forms for each synchronization method:

Figure 7: Synchronization methods in the cdevSystem object

```
class cdevSystem
{
public:
   int flush();
   int poll();
   int pend();
}
```

flush: Flush all pending output to the network for those services that perform send buffering.

poll: Flush all pending output, and process all received replies (including dispatching callbacks, if any). Any received monitor callbacks will be delivered as well.

pend: Flush all pending output, and process all replies for the specified number of seconds. If the time argument is omitted, wait until all replies have been received. If a monitor operation was started, this waits for the connection (first callback) only.

Group Object

An alternative mechanism for waiting for asychronous calls is through the use of groups. A group keeps track off all I/O operations started from the time the group is started until the group is ended. Groups may be nested or overlapped, and support the same flush/poll/pend operations as the system object. For example:

Figure 8: Sample usage of the cdevGroup object

```
cdevGroup g1, g2;
...
g1.start();
mag1.sendCallback(...);
mag2.sendCallback(...);
g2.start();
bpm1.sendNoBlock(...);
bpm2.sendNoBlock(...);
g1.end();
mag3.sendCallback(...);
bpm3.sendNoBlock(...);
g2.end();
g1.pend(4.0);
// at this point operations on mag1, mag2, bpm1, & bpm2 are done
// or have timed out (error handling discussed later).
...
// mag3 and bpm3 may not be finished yet, pend on the second
// group to wait for their completion
g2.pend();
```

Note: Allowing nested or overlapped groups allows library calls to start or stop groups without needing to know if another group is already active.

Groups may be operated in one of two modes. Immediate mode (default) causes grouped operations to be immediately executed, and the group is just used for completion synchronization. Deferred mode causes messages to be held back until the group is ended and the operations are flushed. After the operations are complete, he group of operations may be flushed again (without re-posting), allowing the group to function as a list manager/executor.

Request Object

Occasionally, an I/O operation may need to be repeated may times. In this case, it is not efficient to parse the message each time to determine which server to use. It is possible to bypass this parsing by creating a request object. The request object is like the device object, except that the message string is specified at creation time, and the request object is therefore specific to the particular underlying service to which that message must be directed.

In the case of EPICS channel access, the request object opens and maintains the channel to the EPICS process variable. The request object is NOT the same as an EPICS channel, in that the request object binds a device and a message, and the

message implies a direction. If the message is considered to be of the form *verb + attribute* then the EPICS channel essentially binds device and attribute but not verb. In some future release of CDEV, a more channel like object may be included if there is a demand for it. If two cdevRequestObjects reference the same EPICS channel, only a single channel access connection is established.

The following code demonstrates sending a set of data values to a single magnet using a request object. Note that the send call omits the message argument:

Figure 9: Sample usage of the cdevRequestObject object

```
cdevRequestObject& mag1BDL =
    cdevRequestObject::attachRef("mag1","set BDL");
cdevData mydata;
...
for (i=0;i<100;i++)
    {
    mydata = i*10;                // 10 amp steps
    mag1BDL.send(mydata,NULL);    // ignore errors for now
    sleep(1);
    }
```

In most cases, the device object will create the request object to perform a requested I/O operation. All 3 forms of send are supported by the request object, with a calling syntax identical to the device calls without the message argument. The interface to the request object is given below:

Figure 10: Structure of the cdevRequestObject object

```
class cdevRequestObject
{
public:
    char* message();            // returns the message string
    cdevDevice& device();       // returns the device object
    int state();                // connected/disconnected/
    int access();               // read/write/none
    int send(cdevData& out, cdevData& result);
    int sendNoBlock(cdevData& out, cdevData& result);
    int sendCallback(cdevData& out, cdevCallback callback);
}
```

When an asynchronous operation completes, the relevant request object is passed to the user's callback routine. This allows the user to extract the message string and device name from the request object if necessary or desired.

File Descriptors and "select"

An alternative mechanism for dealing with asynchronous operations is to directly test the file descriptors being used. This interface is currently implemented in the cdevSystem class. The pertinent methods have the following form:

Figure 11: File descriptor routines of the cdevSystem object

```
typedef void (*cdevFdChangedCallback)(int fd, int opened,
                void* userarg)
class cdevSystem
{
public:
    ...
    int getFd(int fd[], int &numFD);
    int addFdChangedCallback(cdevFdChangedCallback cbk, void* arg);
}
```

The getFd method will populate a user allocated array of integers (fd) with the file descriptors currently in use by the cdevSystem object. The number of file descriptors that were allocated by the user is specified in the numFD variable. Upon completion, the numFD variable will be set to the number of file descriptors actually copied into the fd array. An error is returned if the buffer is not sufficient to store the complete set of file descriptors.

The addFdChangedCallback method allows the user to specify a function to be called each time a file descriptor is added (opened=1) or closed (opened=0).

The following code illustrates the usage of the UNIX select call:

Figure 12: Using CDEV file descriptors with the UNIX select function

```
int myFD[5];

void mySelectLoop ( cdevSystem & system )
    {
    int fd[20];
    int numFD = 15, nfds;
    fd_set rfds; // Ready file descriptors
    fd_set afds; // Active file descriptors

    // Copy the file descriptors I am already using to the list
    memcpy(fd, myFD, sizeof(myFD));

    // Get the file descriptors in use by the cdevSystem object
    system.getFd(&fd[5], numFD);

    // Add in the 5 previously defined file descriptors
    numFD += 5;

    // Zero the active file descriptors
    FD_ZERO(&afds);

    // Setup the active file descriptors
    for(int i=0; i<numFD; i++) FD_SET(fd[i], &afds);

    // Get the maximum number of file descriptors
    nfds = FD_SETSIZE;

    while(1) {
        // Copy the active descriptors to the ready descriptors
        memcpy((void *)&rfds, (void *)&afds, sizeof(rfds));

        // Use select to detect activity on the file descriptors
        if(select(nfds, &rfds, 0, 0, 0)<0) exit(1);

        // Iterate through the list of file descriptors
        for(i=0; i<numFD, i++)
            {
            if (FD_ISSET(fd[i], &rfds)
                {
                // Respond to active file descriptor here
                }
            }
        }
    }
```

4. Error Handling

Most CDEV routines return an integer status, with 0=success. Errors may also be automatically printed or routed to a user supplied error handler. Control over this behavior is through the system object:

Figure 13: Error handling mechanisms provided by the cdevSystem object

```
typedef void (*cdevErrorHandler)(int severity, char* text,
                                 cdevRequestObject& request);
class cdevSystem
{
public:
    ...
    int autoErrorOn();
    int autoErrorOff();
    cdevErrorHandler setErrorHandler(cdevErrorHandler handler = 0);
    int reportError(int severity, char *name,
                    cdevRequestObject* request, char* format, ...);
    void setThreshold ( int errorThreshold );
}
```

The *severity* parameter indicates the level of the error. Severity levels are informative, warning, error, and severe error. The following values are defined in cdevErrCode.h to describe the severity of an error message:

Figure 14: Enumerated severity codes generated by CDEV

CDEV_SEVERITY_INFO	**Informative message**
CDEV_SEVERITY_WARN	**Warning message -** operation encountered a problem during processing.
CDEV_SEVERITY_ERROR	**Error message -** operation cannot be completed
CDEV_SEVERITY_SEVERE	**Severe/Fatal Error -** cdev cannot continue execution

autoErrorOn Turn on default error handling, which prints error to stdout.

autoErrorOff Turn off default error handling.

setErrorHandler Set a new function to be the system error handler. Return the old function pointer. Omitting the argument resets the handler to the default handler which simply prints error messages to stderr.

reportError Routine which behaves like printf with 3 additional arguments (severity, caller's name, and request object if available).

Note: For simplicity of use, a very compact set of error codes is implemented in CDEV. The following error codes are defined in the header file cdevErrCode.h.

Figure 15: Enumerated error codes generated by CDEV

```
CDEV_WARNING        = -2  Failure of function is non-consequential
CDEV_ERROR          = -1  Errors that are not in any categories
CDEV_SUCCESS        = 0   Success
CDEV_INVALIDOBJ     = 1   Invalid cdev objects
CDEV_INVALIDARG     = 2   Invalid argument passed to cdev calls
CDEV_INVALIDSVC     = 3   Wrong service during dynamic loading
CDEV_INVALIDOP      = 4   Operation is unsupported (collection)
CDEV_NOTCONNECTED   = 5   Not connected to low network service
CDEV_IOFAILED       = 6   Low level network service IO failed
CDEV_CONFLICT       = 7   Conflicts of data types or tags
CDEV_NOTFOUND       = 8   Cannot find user request (cdevData)
CDEV_TIMEOUT        = 9   Time out
CDEV_CONVERT        = 10  cdevData Conversion error
CDEV_OUTOFRANGE     = 11  Value out of range for device attribute
CDEV_NOACCESS       = 12  Insufficient access to perform request
CDEV_ACCESSCHANGED  = 13  Change in access permission of device
CDEV_DISCONNECTED   = 60  Application has lost server connection
CDEV_RECONNECTED    = 61  Application has regained server connection
```

5. Name Resolution

Some type of name resolution system is needed to (1) locate which package (called a service) underneath CDEV will support the specified message for the specified device and (2) provide parameters needed by the service to contact the server and perform the desired operation. For example, the message "get current" sent to a magnet "mag1" might use the service "ca" (channel access), with parameter "mag1cur.val" (record and field). This parameter is referred to as service data, i.e. data used by the service to perform the operation. The user is typically unaware of this data, which should be supplied by the device implementer.

cdevDirectory Object

Name resolution in CDEV is implemented by the *cdevDirectory* device. The cdevDirectory device supports a *query* message allowing the user to search the Device Directory List (DDL) for devices which are (1) members of a user specified class or (2) match a user specified regular expression.

As with any cdevDevice object, the user instantiates a cdevDirectory device by using the *cdevDevice::attachPtr()* or *cdevDevice::attachRef()* method. The following code fragment illustrates the correct method for attaching to a cdevDirectory device.

Figure 16: Obtaining a reference to the cdevDirectory device

```
cdevDevice & device = cdevDevice::attachRef("cdevDirectory");
```

Like all other devices, the cdevDirectory response to messages. The following messages may be submitted to this device:

- query: Identify the devices that match the selection criteria.
- queryClass: Identify the DDL class from which a device is instantiated.
- queryAttributes: Identify all attributes supported by a device or a DDL class.
- queryMessages: Identify all messages supported by a device or DDL class.
- queryVerbs: Identify all verbs supported by a device or DDL class.
- service: Identify the service that is used by a device/message pair.
- serviceData: Identify service data specified for a device/message pair.
- update: Add information to the cdevDirectory data structure.
- validate: Verify that a device or DDL class contains certain definitions.

By default, there is a single directory device. To allow for arbitrary expansion CDEV will allow each service to register additional directory devices with the system object. A search list of directories may be specified to the system directory device (planned feature).

Each directory acts as if it has a table of information in the following form:

Figure 17: Virtual form of the data in a cdevDirectory

dev-class	dev-name	message	service	service-data
magnet	mag1	on	ca	{pv=M1CSR.VAL default=1}

This table is loaded from a device definition file which describes the mapping from (device,message) to (service,serviceData), where “service” is one of the dynamically loaded CDEV services, and “serviceData” is whatever information that service needs to send the message to the device. In this example the service data contains an EPICS process variable name and a default value to write to that channel.

It is NOT the purpose of the device definition file to completely specify the datatypes of all data passed between device and user. CDEV does considerable data type conversion as needed, and services do reasonable run-time validation. The goal is to make the file as small as is reasonably possible and still maintain readability.

General:

- leading whitespace is not significant
- any amount of whitespace may separate language elements, including space, tab, and newline, with the exception that at least one whitespace character is needed before the colon separating a class name and its inherited classes.
- keywords are case sensitive

NOTE: This specification is preliminary, and will be expanded to support additional features as time allows.

The interface definition consists of three parts: service definitions, device class definitions, and device instantiation (in that order). Standard definitions may be specified via include files using a cpp syntax:

Figure 18: Syntax of the #include directive

```
#include “filename”
```

Service Definition

The service definition declares a service name, and lists all tags which the service will accept for its “serviceData”. The following 3 lines define a service named “myservice” which supports serviceData that specifies values for any of the tags “abc”, “def”, and “ghi”:

Figure 19: Sample service definition

```
service myservice
    {
    tags {abc, def, ghi}
    }
```

As a more concrete example, the following is the specification for the channel access service:

Figure 20: Sample service definition

```
service ca
    {
    tags {pv, default}
    }
```

The “pv” tag is used to specify the process variable name, and the “default” tag is used to specify a default value for a “set” operation (examples are below).

Device Class Definition

Device classes are used to define a collection of similar devices; that is, devices which will respond to the same set of messages. In order to simplify class definition, the following design choices are made:

- multiple inheritance is supported, and
- messages may be defined as verb + attribute, where the list of verbs is defined separately from the list of attributes.

The second item reduces the amount of text needed to define a device, since all operations on a single attribute (get, set, monitor, etc.) may be defined in a single line as long as they all use the same serviceData.

For this initial specification, the class definition contains 4 parts:

- inheritance specification
- verbs
- attributes
- messages

For each attribute or message there are 3 parts:

- attribute or message text
- service name
- service data

Service data is zero or more (name=value) pairs, where the allowed set of names are defined by the “tags” clause in the service specification. The value may contain a pair of angle brackets “<>” into which the device name will be substituted.

Example: Suppose there are a set of magnets for which it is possible to read and write bdl (integral field), and current (amps). That is, the messages “get bdl”, “get current”, “set bdl”, etc. are valid. Also, each magnet responds to the commands “on” and “off”. Further suppose that the beamline position and length of the magnet are available in a static database (assume read/write).

Here is a complete class definition:

Figure 21: Sample inherited class definition

```
class stdio
    {
    verbs {get, set, monitorOn, monitorOff}
    }

class magnet : stdio
    {
    attributes
        {
        bdl ca {pv=<>.bdl};
        current ca {pv=<>.val};
        zpos os {path=<>/phy/z};
        length os {path=<>/phy/len}
        }
    messages
        {
        on  ca {pv=<>CSR.val, default=1};
        off  ca {pv=<>CSR.val, default=0};
        }
    }
```

Note the semicolon at the end of each line of attributes or messages (except for the last line). Note also that the process variable name has a place holder into which the device name will be substituted. More sophisticated name mangling is envisaged, but not for this release. Syntax is still open to revision.

Instance Definition

Instances of a class are given by following the class name with a list of instance names. Device names may be separated by whitespace characters (including newline) or by commas.

Figure 22: Multiple instances of the magnet class

```
magnet : m1 m2 m3;
```

Combined with the previous definition, the message “on” sent to magnet m2 would select the process variable “m2CSR.val” with “1” as the value to write.

Aliases

Sometimes it is convenient to refer to a device by more than one name. The DDL syntax supports simple aliases, one per line.

Figure 23: Alias device name

```
alias myname m1
```

6. Channel Access Service

EPICS / Channel Access version 3.12 is recommended.

Features:

- Supports synchronous and asynchronous send's.
- Tracks file descriptor registration.
- Fetches data in native type (type conversion done on client and not server).
- Provides a default name service so that if a device is not defined, then an EPICS channel access connection is attempted with the record name set equal to the device name, and the field name set equal to the attribute name.

Restrictions:

- Only supports set/get/monitorOn/monitorOff verbs and the pv and readonly tags. Support for arbitrary messages with default data will be in the next release.
- Discards all exception callbacks from channel access in this version.
- Channel access security is not supported in this release.

Compilation options:

- _CA_SYNC_CONN = perform connections synchronously, waiting up to 4 seconds.
- _CDEV_DEBUG = print verbose messages
- _EPICS_3_12 = if not defined, use 3.11 calls instead.