CDEV C Binding Guide

Chip Watson, Jie Chen, Danjin Wu, Walt Akers

Version 1.4 October 24, 1996

TJNAF - Thomas Jefferson National Accelerator Facility

DOCUMENT DATE: Table of contents generated: October 24, 1996 3:12 pm

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

cdev C Binding

Updated: 10/24/96

This working document describes a C binding to the cdev control system interface (see the cdev Design Document, and the cdev User's Guide).

Design Choices:

1. All C routine names begin with "cdev" (common device).
2. Most C routines return completion status; 0==success (allows enumerated errors). A few return integer values (prefixed in this document with "int").
3. All data passed through the interface will be in the form of integers, strings or a self describing structure. Library calls will be provided to manipulate this structure.

Comparison with C++ binding:

1. cdevData, cdevRequestObject, and cdevGroup objects will be accessed through object id's.
2. cdevDevice will only be referenced by device name.
3. Only a single, default, cdevSystem will be supported.
4. Since no function overloading is available, I/O operations on a cdevData object will have to pass a datatype argument, similar to EPICS channel access.
5. Routines which return strings will malloc the string, and the caller must free it.
6. Function names are generally constructed from the class name followed by the method name, e.g. cdevGroupStart.
7. Creating objects will be done by routines like cdevGroupAllocate, which will essentially return a pointer to the created object.
8. cdevSystem methods will drop "System" from the name, and the notion of "system" will be invisible.
9. cdevDevice methods will drop "Device". cdevRequestObject methods will use the shorter "cdevRequest".

C Binding User's Guide

Simple message passing routines:

Devices are addressed by name, and operations are performed by sending a message string and an additional optional data object identifier. This data object is self describing in the sense that it contains indicators of the data type (int, float, etc.) and size (array lengths). Data returned from the control system will also be through a data object.

status = cdevSend (char* devName, char* message,

cdev_data_t out, cdev_data_t in);

The data objects act as small databases able to hold multiple tagged data items. Items are inserted into or extracted from the database via access routines. These access routines are also responsible for any necessary data type conversions. Tags are passed to the cdevData routines as integers, and are looked up in the tag database by specifying an ascii tag name. The most common tags are “value”, “timeStamp”, “status”, and “severity”.

Data object manipulation routines:

cdevDataAllocate (cdev_data_t* id);

cdevDataFree (cdev_data_t id);

cdevDataTagC2I (char* tag, int *tagid);

cdevDataTagI2C (int tagid, char tag);/* caller must free string */**

cdevDataInsertTag (int tag, char* ctag);

cdevDataInsert (cdev_data_t id, int tagid, int type, void * data);

int cdevDataInsertArray(cdev_data_t id, int tagid, int type, void *data, size_t len, size_t ndim);

cdevDataGet (cdev_data_t id, int tagid, int type, void * data);

cdevDataFind (cdev_data_t id, int tagid, void ** data);

int cdevDataGetType (cdev_data_t id, int tagid);

int cdevDataGetDim (cdev_data_t id, int tagid);

int cdevDataGetElems (cdev_data_t id, int tagid);

cdevDataGetBounds (cdev_data_t id, int tagid, int *bounds, int bsize);

cdevDataSetBounds (cdev_data_t id, int tagid, int *bounds, int bsize);

void cdevDataRemoveAll(cdev_data_t id);

cdevDataRemove (cdev_data_t id, int tagid);

cdevDataChangeTag (cdev_data_t id, int tagid, int newtagid);

cdevDataAsciiDump (cdev_data_t id, FILE *fp);

void cdevDataCopy(cdev_data_t from, cdev_data_t *to);

For example, to read the current from a magnet named NLQ1:

cdev_data_t result;

```
float current;
int status, valtag;
status = cdevDataAllocate (&result);
status = cdevDataTagC2I ("value",&valtag);
status = cdevSend("NLQ1","read current",NULL,result);
status = cdevDataGet(result, valtag, CDEV_FLOAT, current);
```

The first call to a device automatically connects to the requested service, initializing any underlying packages as needed. These routines are synchronous (operation completes before return); asynchronous routines are discussed below.

The message (second) argument selects what operation and attribute to access. Many messages are by convention of the form “verb attribute”, and typical verbs include get, set, monitorOn, and monitorOff.

Binding a device and message

In order to avoid the overhead of parsing the device name and message on each call, it is possible to get a handle which binds the device and message into a request.

status = cdevRequestAllocate(char* device, char*message, CDEVREQUESTID* reqid);

Operations may be performed on this request object without specifying device and message.

status = cdevRequestSend(reqid, out, result);

Request objects maintain a connection to a server, and the state of this connection may be obtained:

state = cdevRequestState (reqid);

The current access rights (read, write, none) for this request object may be obtained:

access = cdevRequestAccess (reqid);

Given the request object, it is also possible to extract the device name and message:

cdevRequestDevice (reqid, char name); /* caller must free string */**

cdevRequestMessage (reqid, char msg);**

Monitoring and Asynchronous I/O

A message sent to a device may result in an asynchronous reply, or more than one reply. In addition, abnormal conditions may occur in the device which are of interest to the application. Each of these results in a message back to the client which is asynchronous with respect to program execution. Several calls are provided to deal with asynchronous data.

Asynchronous Messages. An asynchronous version of the message send call will send the message, but not wait for the reply. Two async forms are supported: one returning data to a caller’s data object, and the other returning data to a callback function. The reply argument is replaced by an additional structure which specified a user callback function.

status = cdevSendNoBlock(char* device, char* message,

cdev_data_t out, cdev_data_t result);

status = cdevSendCallback(char* device, char* message,

cdev_data_t out, CDEVCALLBACK* callback);

Similarly for request objects:

status = cdevRequestSendNoBlock(cdev_data_t out, cdev_data_t result);

status = cdevRequestSendCallback(cdev_data_t out, CDEVCALLBACK* callback);

Monitoring: Monitoring is started and stopped by two using the message "monitorOn xxx" and "monitorOff xxx".

status = cdevSendCallback("deviceName", "monitorOn attributeName",NULL, callback);

Synchronization. In order to synchronize with asynchronous messages, both a poll and a pend call are available, as well as an explicit flush. A pend with 0.0 seconds waits forever.

cdevFlush ();

cdevPoll ();

cdevPend (float seconds);

If seconds==0.0, wait forever.

Grouping. Grouping of operations will be similar to that specified for EZCA, with slight variations. The grouping calls are useful for synchronizing with a set of asynchronous calls.

cdevGroupAllocate(GROUPID* groupid);

cdevGroupStart(groupid);

cdevGroupEnd(groupid);

cdevGroupPoll(groupid);

cdevGroupPend(groupid);

cdevGroupAllFinished (groupid);

cdevGroupStatus (groupid, int *status, int *nstatus); /* initially nstatus = len of status */

File Descriptors. To integrate other asynchronous systems, such as X windows, it is often necessary to obtain file descriptors for select operations.

status = cdevGetFD(int fd[], int * numFD);

Context. Operations on a device take place within a context. The context is maintained by the device as a cdev_data_t structure. The following routines are used to get and set the context; setting items within the context is done by the cdevData* routines.

cdevGetContext(char *devname, cdev_data_t *id);

cdevSetContext(char *devname, cdev_data_t id);

Private Data. Each device also maintains a private data pointer, to be used by the application developer for any purpose desired.

cdevGetPrivate (char *devname, void **data);

cdevSetPrivate (char *devname, void *data);

These same four routines are available for request objects:

cdevRequestGetContext (reqid, cdev_data_t *id);

cdevRequestSetContext (reqid, cdev_data_t id);

cdevRequestGetPrivate (reqid, void **data);

cdevRequestSetPrivate (reqid, void *data);

Error Reporting and Handling

This group of routines is also modeled after EZCA (assume identical functionality).

Turn automatic error reporting on and off:

cdevAutoErrorOn ();

cdevAutoErrorOff ();

Set the severity threshold at which errors should be reported:

cdevSetThreshold (int new);

cdevReportError (int severity, char *name,
CDEVREQUESTID request, char* format, ...);

cdevGetErrorString ();

cdevSetErrorHandler (function);

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.

CDEV_SUCCESS = 0	Success
CDEV_INVALIDOBJ= 1	Invalid cdev object used
CDEV_INVALIDARG= 2	Invalid argument passed to cdev method
CDEV_INVALIDSVC= 3	Wrong service during dynamic loading
CDEV_NOTCONNECTED= 4	Not connected to low-level network service
CDEV_IOFAILED = 5	Low-level network service IO failed
CDEV_CONFLICT = 6	Conflicting of data types or data tags
CDEV_NOTFOUND = 7	Cdev cannot find specified data or tag
CDEV_TIMEOUT = 8	Time out
CDEV_CONVERT = 9	cdevData conversion error

Name Services

There is a special device named cdevDirectory which provides a set of query capabilities:

status = cdevSend(“cdevDirectory”, ”query”,
cdev_data_t selection, cdev_data_t result);

The selection data object specifies one or more tags used to select device names. The result contains a list of devices accessed through the “value” tag.