A CDEV Tutorial

Techniques for using 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 11:46 am

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. Developing CDEV Applications

Overview

The purpose of this section is to describe the steps that a CDEV user must perform in order to effectively develop a CDEV application. This section assumes that the user is either using one of the CDEV services provided in the CDEV distribution, or has a local or third party developed CDEV service with which the application will communicate.

Steps in Developing a CDEV Application

1. **Construct a CDEV device definition file.** The CDEV device definition file is the roadmap that CDEV uses to define services and determine which device and message combinations are associated with those services. This file should be constructed and defined in the ***CDEVDDL*** environment variable prior to executing a CDEV application. Check with the CDEV adminstrator to determine if this file already exists on your system.

2. **Design and write your source code.** This step is intentional broad in scope because the capabilities provided by various CDEV services may be completely dissimilar in nature. The primary capability that CDEV provides is standardization of the interface between any application and any underlying CDEV service.

3. **Compile and link application with the CDEV library.** This step requires the source code to include the necessary CDEV C++ header files and linking with the **libcdev.sl** or **libcdev.a** libraries.

4. **Set any required environment variables.** The primary environment variables required by CDEV are **CDEVSHOBJ** and **CDEVDDL**. CDEVSHOBJ is used to located shared objects, and cdevddl is used to specify the complete path to the CDEV device definition file. Individual services may have additional requirements.

5. **Execute your application.**

Note: Applications linked with the archive library are not capable of loading dynamically loadable CDEV services.

2. Developing Device Definition Language Files

Device Definition Language (DDL) File Overview

The Device Definition Language File or DDL File is the roadmap that CDEV uses to associate certain device and message combinations with a CDEV service. The DDL file provides the mechanism for defining *service definitions, class definitions* and *instance definitions.*

Service Definitions

The service definition is used to publicly define the existence of a CDEV service. The service definition declares the name of the service and the names of any special cdevData tags that the service will use. The following is a simple service definition.

Figure 1: Service definitions in the CDEV DDL file

```
/*
 * Definition of service X
 */
service x
    {
    tags { tag1, tag2 }
    }

/*
 * Definition of service y
 */
service y
    {
    tags {}
    }
```

"service" Keyword

Service Name

List of Tags

"tags" Keyword

The service definition above defines the service "x" and service "y". During the execution of a CDEV application, when a device and message combination is related to the "x" service, CDEV will attempt to load the dynamic library for the service. By convention the name of the library will be "xService.so", where "x" is the name of the service within the DDL file.

The service definition also defines the tags "tag1" and "tag2" in conjunction with service "x". These tags will be used later to define special service data that is related to a specific device attribute. Service "y" has no special tags associated with it.

This example also illustrates the formats of comments within the DDL file. The syntax of the DDL file accommodates the use of C style multi-line comments.

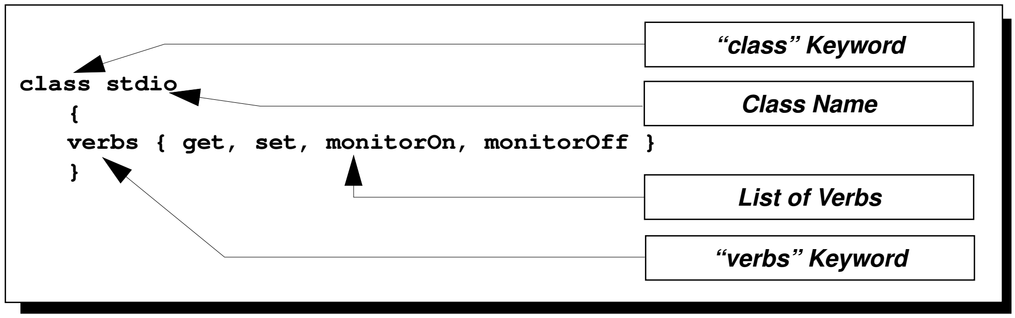
Class Definitions

The class definition is used within the DDL file to declare a collection of *verbs*, *attributes* or *messages* that may be applied to one or more CDEV device instances.

Verb Definitions

The following examples shows the syntax for declaring a class definition that contains only a list of verbs.

Figure 2: CDEV DDL class definition containing a list of verbs



In the above example, the "*stdio*" class is defined and it is declared to contain four verbs: *get, set, monitorOn* and *monitorOff.* Any other class that is derived from this class, or any device that is instantiated from this class will inherit these four verbs.

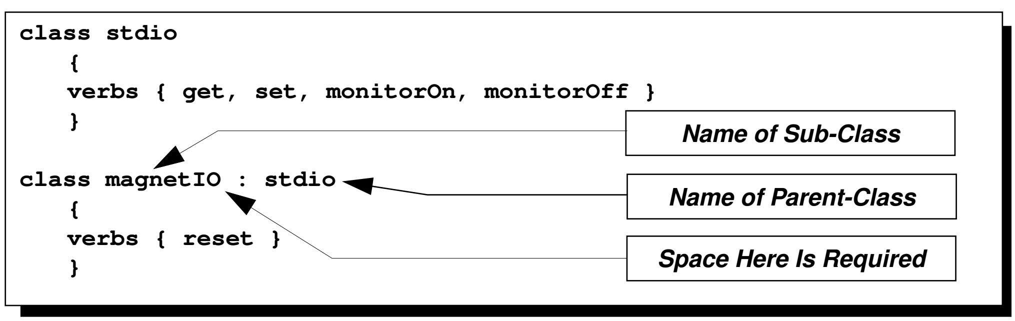
Class Inheritance

The CDEV DDL file supports the notion of inheriting one class from another class. The pre-existing class is called the *super-class*, while the class that is inheriting from the pre-existing class is called the *sub-class*. To specify inheritance within the CDEV DDL file, the syntax is similar to the syntax used in C++. The name of the new class is followed by a colon, which is followed by the name of the *super-class*.

In order to support the possibility of a colon as a character in the class name, a space must separate the name of the class and the colon.

In the next example, the "magnetIO" class is defined. This class inherits from the *"stdio"* class, and receives the verbs *get, set, monitorOn, and monitorOff.* To this list of verbs, it adds a new verb: *"reset"*.

Figure 3: CDEV DDL class definition using inheritance



Attribute Definitions

Class definitions are also used to define *attributes*. An attribute is a component of a CDEV device that can be combined with a *verb* to define a message. When an *attribute* is declared within a class, CDEV will assume that the *attribute* supports each of the *verbs* that exist within the class.

Attribute Service Data

When the attribute is defined, the user must define the name of the service that will receive the messages associated with it, and any *service data* that is associated with it. This service data is defined using the tag names that are declared in the *tags* section of the service definition.

The following example declares the class *“element”.* This class inherits its verbs from the *“stdio”* class. Each attribute is associated with a *service* name and is followed by service data.

Figure 4: CDEV DDL class definition containing a list of attributes

```
/*****************************
 * Definition of service X
 *****************************/
service x
   {
   tags { tag1, tag2 }
   }

/*****************************
 * Definition of stdio class
 *****************************/
class stdio
   {
   verbs { get, set, monitorOn, monitorOff }
   }

/*****************************
 * Definition of element class
 *****************************/
class element : stdio
   {
   attributes
      {
      value1 x {tag1=1.0, tag2=Value};
      }
   }
```

“attributes” Keyword

Attribute Name

Semi-Colon Line Terminator

Special Service Data

Name of Associated Service

Service data also supports the notion of expression replacement. If the character string ***“<>”*** appears in the service data, it will be replaced with the instance name of the device that is instantiated from this class.

Message Definitions

A message definition can be declared within a CDEV DDL class definition to define one word atomic commands. Unlike attributes and verbs, a message is a stand-alone instruction that can be sent to a device.

The structure of a *message* definition is similar to the *attribute* definition. The entries in this section contain the name of the service and special service data that is associated with the message.

The following example shows a class that contains a single message definition. Note that the class does not define any verbs or attributes.

Figure 5: CDEV DDL class definition containing a list of messages

```
/*****************************
 * Definition of service X
 *****************************/
service x
   {
   tags { tag1, tag2 }
   }

/*****************************
 * Definition of element class
 *****************************/
class element
   {
   messages
      {
      on x {tag1=<>.Val, tag2=Text};
      }
   }
```

"messages" Keyword

Message Name

Device Name Placeholder

Service Name

Device Instances

The *device instances* section is where devices are created from class definitions. Each device that is instantiated from a class inherits all of the verbs, attributes and messages from that class, as well as from each class that it inherits from.

The syntax for instantiating a device is as follows: the name of the class, followed by a colon, followed by the list of device names that are being defined. The following figure shows the syntax for defining a list of devices of the *"element"* class.

Figure 6: CDEV DDL class definition to instantiate a list of devices

```
/*****************************
 * Instantiation of devices
 *****************************/

element :
   device1
   device2
   device3;
```

Class Name to Instanciate

Colon Separator

List of Devices

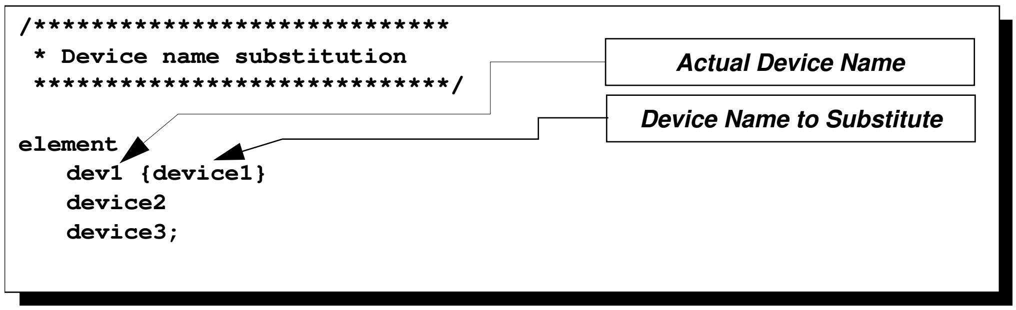
Semi-Colon List Terminator

Device Name Substitution

As described earlier, angle braces “<>” in the service data definition for an attribute will be replaced with the name of its associated device. A name substitution technique is available that allows the DDL writer to specify that a different device name should be inserted. The substitution device name is specified in conjunction with the actual device name when the device is instanciated.

The following example illustrates the syntax for instantiating a device and specifing a substitute name for it. In this example, the string “*device1*” will be substituted wherever the string *“dev1”* would normally be used.

Figure 7: Implementing device name substitution in a CDEV DDL file

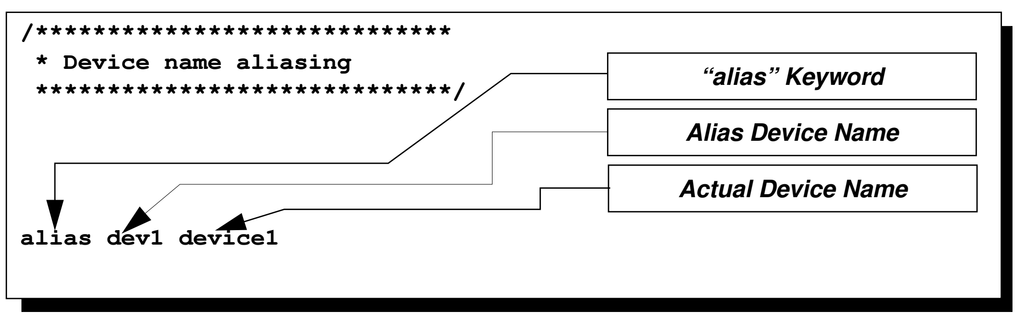


Device Name Aliasing

Device name aliasing is a more powerful alternative to device name substitution. A device’s alias may be used in applications to obtain and communicate with its corresponding device. Device name aliases are specified in the DDL file on an individual basis by using the *alias* keyword.

The syntax for adding an alias to a device is illustrated in the example below. In this example, the alias *“dev1”* can be used interchangably with the actual device name *“device1”.*

Figure 8: Implementing device name aliasing in a CDEV DDL file



Defining cdevCollections

A cdevCollection device can be implemented in one of two ways; the developer can create an empty cdevCollection dynamically and add devices to it, or a collection entry can be placed in the CDEV DDL file that identifies the collection and its constituent devices. The syntax for defining a cdevCollection device in the CDEV DDL file is as follows.

Figure 9: cdevCollection Definition in the CDEV DDL File

```
collection cDevice1 :
device0,
device1,
device2
;

collection cDevice2 :
device3
device4
device5
;
```

In the example above two cdevCollections are defined. A cdevCollection definition is started by using the keyword "***collection***", followed by a space and then the name of the collection. Note that the name of the collection must be unique and cannot be shared with any other device in the CDEV system.

The device name is followed by a white space character (***which is required***), and then a colon, followed by another white space character and then the list of devices. The device names that are in the list may be separated by either white space of a ***comma*** delimiter. The list of devices is terminated by a ***semicolon***.

The current implementation of the CDEV DDL is order dependent. Because of this, the developer should place the collection definitions after the definitions of all devices and aliases.

#include Directive

The *#include* directive is supported within the CDEV DDL file syntax. This directive is used to include the contents of another DDL file within the current DDL file. The syntax is the same as its C language counterpart.

Figure 10: Using the #include directive in a CDEV DDL file

```
#include "filename.ddl"
```

Sample DDL

The following sample DDL file demonstrates all current features of the Device Definition Language.

Figure 11: Sample complete DDL file

```
/*****************************************************************
 * Service Definitions:
 * This section contains the definitions for individual services.
 *****************************************************************/

/*****************************************************************
 * ca Service (channel access)
 * The library caService.so will be loaded and called whenever a
 * call to this service needs to be executed.
 *
 * This services support the following special tags:
 *   PV       = Process Variable: this tag is used to specify the
 *              actual name of the attribute that is being
 *              defined.
 *   READONLY = This is a logical tag that indicates if the
 *              attribute is readonly (1) or read/write (0).
 *****************************************************************/
service ca {
   tags { PV, READONLY }
   }

/*****************************************************************
 * Class Definitions
 * This section contains the definition for individual classes.
 *****************************************************************/

/*****************************************************************
 * stdio Class:
 * This class defines the basic set of verbs that will be
 * supported by all other classes in this DDL file.  The verbs
 * declared in this class are:
 *     get        = Get the value of an attribute of a device
 *     set        = Set the value of an attribute of a device
 *     monitorOn  = Monitor changes of an attribute’s value
 *     monitorOff = End monitoring changes of an attribute’s value
 *****************************************************************/
class stdio {
   verbs { get, set, monitorOn, monitorOff }
   }
```

Figure 11: Sample complete DDL file (continued)

```
/*******************************************************************
 * magnet Class:
 * This class inherits its list of verbs from the stdio class.  It
 * then adds a list of attributes that these verbs may operate
 * on.  The following attributes are specified.
 *
 *    current  : The current attribute will be serviced by ca.
 *    bdl      : The bdl attribute will be serviced by ca.
 *    length   : The length attribute will be serviced by ca.
 *
 * This class also supports the messages "on" and "off".  For
 * illustration purposes, these messages will operate on the val
 * field, and therefore, have the same service data.
 *******************************************************************/
class magnet : stdio
   {
   attributes {
      current ca {PV=<>.VAL, READONLY=0};
      bdl     ca {PV=BDL_<>.VAL, READONLY=0};
      length  ca {PV=LEN_<>.VAL, READONLY=1};
      }
   messages {
      on  ca {PV=<>.VAL, READONLY=0};
      off ca {PV=<>.VAL, READONLY=0};
      }
   }

/*******************************************************************
 * Device Instances:
 * This section contains the instantiations of individual devices.
 *******************************************************************/

/*******************************************************************
 * The following is a list of devices that are memebers of the
 * magnet class.  That means that each of these devices supports
 * all of the attributes, verbs and messages that are defined by
 * the magnet class and the stdio class that it inherits from.
 *******************************************************************/
magnet :
   MQB1S01
   MQB1S02
   MQB1S03
   MQB1S04

/*******************************************************************
 * End of Device Definition File
 *******************************************************************/
```

3. Using the *cdevSystem* Object

Overview of the cdevSystem Class

The cdevSystem C++ class can be described as a communications junction between a CDEV application and CDEV services. The cdevSystem object is responsible for interpreting the device definition file, dynamically loading CDEV services upon request, detecting and responding to I/O events, and providing a mechanism for polling services for updates.

Currently, only the default cdevSystem object may be used within an application. Ultimately, multiple cdevSystem objects may be created, and each of them will maintain individual copies of CDEV services and devices within them.

Most CDEV classes that are accessed directly by applications have static factories (special constructors). Any CDEV object that is constructed by one of these factories is automatically a member of the *default* cdevSystem.

Public Functions of the cdevSystem Class

attachRef

*static cdevSystem& attachRef(char *name, char *prefix);*

Obtains a reference to a new or existing cdevSystem object that has the specified name. The *prefix* parameter is optional, but, if provided will be prepended to the name of each *device* that is requested through the named cdevSystem object before it is found in the device definition file.

attachPtr

static cdevSystem attachPtr(char *name, char *prefix);*

Obtains a pointer to a new or existing cdevSystem object that has the specified name. The *prefix* parameter is optional, but, if provided will be prepended to the name of each *device* that is requested through the named cdevSystem object before it is found in the device definition file.

defaultSystem

static cdevSystem &defaultSystem (void);

Obtains a reference to the default cdevSystem object. In most applications this mechanism is used to access the cdevSystem object.

getDevice

cdevDevice getDevice (char *device);*

Retrieves a pointer to a cdevDevice object by name. The object that is returned by this function will be owned and controlled by the cdevSystem object that created it. This means that *flush, poll* and *pend* commands must be routed through the owning cdevSystem object in order to effect this cdevDevice object.

getRequestObject

*int getRequestObject (char *device, char *msg,*
cdevRequestObject &req);

Retrieves a reference to a cdevRequestObject from a specified device name and message name. The object that is produced by this function will be owned and controlled by the cdevSystem object that created it. This means that *flush, poll* and *pend* commands must be routed through the owning cdevSystem object in order to effect this cdevRequestObject. Returns *CDEV_SUCCESS* or an enumerated error code.

name *char *name (void);*

Returns the name of this cdevSystem object.

prefix *char *prefix (void);*

Retrieves a pointer to the prefix string that is currently being used by the cdevSystem object. See the *attachPtr* entry for a description of the prefix string.

prefix *void prefix (char *pre);*

Changes the prefix string that is being used by the cdevSystem object. See the *attachPtr* entry for a description of the prefix string.

flush *int flush (void);*

Flushes any pending outbound requests to their respective services.

poll *int poll (void);*

Directly polls each of the cdevSystem's underlying services for activity, and delivers any asynchronous callbacks that are ready.

pend *int pend (int fd);*

Waits for a default period of time for the specified file descriptor to have an I/O event. If the fd parameter is not provided, the cdevSystem object will wait for an I/O event on any of its contained file descriptors. When an event occurs on one of the file descriptors, the cdevSystem object will call the respective cdevService to process the event and dispatch any asynchronous callbacks that are ready.

pend *int pend (double seconds, int fd);*

Processes all I/O events that occur on the file descriptor during the specified period of time. If the fd parameter is not provided, the cdevSystem object will wait for I/O events on all of its contained file descriptors. When an event occurs on one of the file descriptors, the cdevSystem object will call the respective cdevService to process the event and dispatch any asynchronous callbacks that are ready.

getFd *int getFd (int fd[], int &numFD);*

Retrieves a list of file descriptors that are contained within the cdevSystem object. The fd array must be preallocated, and the maximum number of elements should be specified in the numFD parameter. Upon completion, the fd array will be populated with the list of file descriptors, and the numFD parameter will contain the actual count.

addFdChangedCallback

*int addFdChangedCallback (cdevFdChangedCallback cbk, void*userarg)*

Adds a new function to be called each time a service announces to the system that it has opened or closed a file descriptor. The function will be called with 3 arguments:

*void (*cdevFdChangedCallback) (int fd, int opened, void *userarg)*

The first argument specifies the file descriptor, the second is 1 for opened, 0 for closed, and the last argument is the user specified argument.

autoErrorOn *int autoErrorOn (void);*

Informs the cdevSystem object that it should use its internal default error handler to process any error messages that are generated by objects within its control. This is the default operating condition for the cdevSystem object.

autoErrorOff *int autoErrorOff (void);*

Informs the cdevSystem object that it should use the user supplied error handling function to process any error messages that are generated by objects within its control.

reportError *int reportError (int severity, char *name, cdevRequestObject *obj, char *formatString,...);*

Emits an error message. The severity field indicates the *severity* of the error, the *name* string identifies the object that generated the error, the *obj* parameter is the cdevRequestObject that was in use when the error occurred, and the *formatString* and additional parameters (...) should be formatted in the same manner as the parameters used by printf.

The integer used by severity should have one of the following values indicating the severity of the error that has occurred.

CDEV_SEVERITY_INFO: No error.

CDEV_SEVERITY_WARN: An error occurred that should not impact continued processing.

CDEV_SEVERITY_ERROR: An error occurred and should be corrected before continuing.

CDEV_SEVERITY_SEVERE: A severe or fatal error has occurred and CDEV cannot continue normal execution.

setErrorHandler *void setErrorHandler (cdevErrorHandler handler);*

Used to to install a user specified error handler. This error handler will be called when an error occurs if the *autoErrorOff* method has been used to disable the default error handler. The user provided error handler should have the following prototype:

*void handler (int severity, char *text, cdevRequestObject *obj);*

The *severity* parameter will contain one of the integers specified in the *reportError* documentation, the *text* parameter will contain the text of the error, and the *obj* parameter will contain the cdevRequestObject that was in use when the error occurred.

setThreshold *void setThreshold (int errorThreshold);*

Used to specify the level of severity at which errors should be submitted to the error handler. The value of *errorThreshold* should be one of the severity levels specified in the *reportError* method.

Sample Code

The following sample application shows the use of the three methods for obtain a cdevSystem object. For examples of how to use the flush, poll and pend methods, see the documentation for the cdevDevice object.

Figure 12: Methods for obtaining a cdevSystem object

```
#include <cdevSystem.h>

void main()
   {
   // ************************************************************
   // * The first and most common approach is to directly request
   // *a reference to the default cdevSystem.
   // ************************************************************
   cdevSystem & default = cdevSystem::defaultSystem();

   // ************************************************************
   // * Get a pointer to the name of the default cdevSystem.
   // ************************************************************
   char * sysName = default.name();

   // ************************************************************
   // * Use the attachRef method to obtain a reference to the
   // * default cdevSystem by name.
   // ************************************************************
   cdevSystem & default2 = cdevSystem::attachRef(sysName);

   // ************************************************************
   // * Finally, use the attachPtr method to obtain a pointer to
   // * the default cdevSystem by name.  Additionally, set the
   // * prefix for the default system to "TEST".
   // ************************************************************
   cdevSystem * default3 = cdevSystem::attachPtr(sysName, "TEST");

   // ************************************************************
   // * Output the name and prefix for all three systems.
   // ************************************************************
   printf("%s:%s\n", default.name(), default.prefix());
   printf("%s:%s\n", default1.name(), default1.prefix());
   printf("%s:%s\n", default2->name(), default2->prefix());
   }
```

The following sample application illustrates how to install a user specified error handler into the default cdevSystem object.

Figure 13: Installing a user specified error handler

```
#include <cdevSystem.h>

// ****************************************************************
// * This the default error handler.  It simply outputs the error.
// ****************************************************************
void myHandler(int severity, char *text, cdevRequestObject &obj)
    {
    if(severity>0)
        {
        fprintf(stderr, "ERROR (%i): %s\n", severity, text);
        }
    else fprintf(INFORMATION: %s\n", text);
    }

void main()
    {
    // ************************************************************
    // * Obtain a reference to the default cdevSystem.
    // ************************************************************
    cdevSystem & default = cdevSystem::defaultSystem();

    // ************************************************************
    // * Call the setErrorHandler method to install the new
    // * errorHandler.
    // ************************************************************
    default.setErrorHandler(myHandler);

    // ************************************************************
    // * Turn off auto error handling to tell the system to use
    // * the new user specified error handing function.
    // ************************************************************
    default.autoErrorOff();
    }
```

4. Using the *cdevDevice* Object

Overview of the cdevDevice Class

The cdevDevice C++ class is the user's primary interface to CDEV. In fact, it is possible to develop complete CDEV applications using only cdevDevice and cdevData objects. The cdevDevice class defines the basic interface that all applications must use to transmit messages to a CDEV service.

A cdevDevice may be created using the static member functions ***attachPtr*** or ***attachRef***. If one of these methods is used, the device will be created within the context of the default *cdevSystem*. If the application wishes to create a device within a cdevSystem other than the default, it must use the ***getDevice*** method of the desired cdevSystem object.

A cdevDevice object is bound by name to a specific device when it is instantiated. However, the underlying service is not selected until a message is specified with a ***send*** command. Therefore, simply creating a device does not ensure that the device name is valid, nor that it supports a given message.

Messages may be sent to a device using one of the three send member functions, these are: ***send, sendNoBlock****, and* ***sendCallback***. The syntax and functionality of these methods is described in the section below.

Input and output to the specified send methods is managed through the use of cdevData objects. Unlike the cdevDevice object which is bound to a specific system, the cdevData object is completely independent. These objects store the data values (called *properties*) that are sent to and received from devices. The application may indicate the *properties* that it wishes to receive as output from the cdevDevice object by specifying them in a special cdevData object called the *context*.

Examples at the end of this section will illustrate the correct ways to implement these operations.

Public Functions of the cdevDevice Class

attachRef *static cdevDevice& attachRef (char *name);*

Obtains a reference to a cdevDevice object by name. By default, the new object will be managed by the default cdevSystem.

attachPtr *static cdevDevice* attachPtr (char *name);*

Obtains a pointer to a cdevDevice object specified by name. By default, the new object will be managed by the default cdevSystem.

detach *static void detach (cdevDevice& dev);*

Removes a referenced cdevDevice object from its associated cdevSystem object. Ordinary applications should never use this command.

detach *static void detach (cdevDevice* dev);*

Detaches the cdevDevice object specified by dev from its associated cdevSystem object. Ordinary applications should never use this command.

getRequestObject *cdevRequestObject* getRequestObject (char *msg);*

Obtains a service specific request object. This function uses the name of the device specified within the cdevDevice object, and the message provided by the caller to determine which CDEV service will be used to service this request. The service is then loaded (if necessary), and is contacted to provided a request object for the device/message combination. This cdevRequestObject will then be used to communicate with the service directly.

name *const char *name (void) const;*

Returns the name of the device.

system *cdevSystem& system (void) const;*

Obtains a reference to the underlying cdevSystem object that manages this device.

setContext *int setContext (cdevData& cxt);*

Used to insert a cdevData object containing tagged values that control optional behavior of the underlying device. The context is often used to specify which properties (value, status, severity) a device returns in response to a “get” message. It is important to note that any change made to the context of a *cdevDevice* will automatically be propagated to all *cdevRequestObjects* created from that device. Therefore, any message specific context settings should be specified directly to the cdevRequestObject, rather than to the cdevDevice.

getContext *cdevData & getContext (void);*

Retrieves a reference to the cdevData object that contains the context for a specific device.

setPrivate *void setPrivate (void * data);*

Associates a user specified data object with this cdevDevice object. The pointer can be retrieved later using the *getPrivate* method.

getPrivate *void * getPrivate (void);*

Retrieves a pointer to a data object that was placed in this cdevDevice object using the setPrivate function.

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

The send function is the standard method for synchronously communicating with a device. The caller provides the message as a character string in the *msg* parameter. The *out* cdevData object contains any property values that the device will need to perform the task. The *result* cdevData object will contain the output properties that resulted from the call. This function will return one of the error code defined in cdevErrCode.h.

Figure 14: Return codes generated by the send method

```
CDEV_SUCCESS:        The message was processed successfully.
CDEV_ERROR:          Failed to process message.
CDEV_INVALIDOBJ:     Invalid CDEV request object used.
CDEV_INVALIDARG:     Invalid argument passed to CDEV call.
CDEV_INVALIDSVC:     Wrong service during dynamic loading.
CDEV_NOTCONNECTED:   Not connected to low level network
                     service.
CDEV_IOFAILED:       Low level network service IO failed.
CDEV_CONFLICT:       Conflicts of data types or tags.
CDEV_NOTFOUND:       Cannot find specified data in cdevData.
CDEV_TIMEOUT:        Time out.
CDEV_CONVERT:        cdevData conversion error.
CDEV_OUTOFRANGE:     Value out of range for device attribute.
CDEV_NOACCESS:       Insufficient access to perform request.
CDEV_ACCESSCHANGED:  Change in access permission of device.
CDEV_DISCONNECTED:   The service has lost contact with the
                     device.
CDEV_RECONNECTED:    The service has regained contact with the
                     device.
```

sendNoBlock

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

The *sendNoBlock* method uses the same parameters and syntax as the *send* method. However, rather than waiting for the underlying service to respond to the request, this function will return immediately. The caller must utilize some synchronization method such as a cdevGroup object to detect when this transaction has been completed, or test for new data within the result data object.

sendCallback

*int sendCallback (char *msg, cdevData &out,*
cdevCallback &callback);
*int sendCallback (char *msg, cdevData *out,*
cdevCallback &callback);

The sendCallback function is the standard method for asynchronously communicating with a device. Rather than providing a result cdevData object, this method requires the user to provide the address of a cdevCallback object. This object contains a user supplied pointer and the address of a function to call when the message has been successfully processed.

Sample Code

The following sample applications illustrate many of the uses of the cdevDevice object. Additionally, these examples show in depth usage of the cdevData object and the concept of context.

Attaching to a cdevDevice Object

Within a cdevSystem object, a cdevDevice is created only once for any specified device name. After that, all requests for that device are given a pointer to the same cdevDevice object. This technique is called *attaching* to a device.

There are two approaches to attaching to a cdevDevice; through a specific cdevSystem object, or through the cdevDevice interface. Attaching to the device using the cdevSystem object has the benefit of making it possible to manage the device through a system other than the default system. Using the cdevDevice interface directly always places the new device in the default system, but has the benefit of reducing the complexity of the source code by hiding the use of the cdevSystem object.

The next example illustrates how to attach to a cdevDevice using the *attachRef* and the *attachPtr* methods. The *attachRef* method allows the caller to obtain a reference to the cdevDevice object, while the *attachPtr* method allows the user to obtain a pointer to the same object. This example also demonstrates the long hand approach for retrieving a cdevDevice pointer from the system object.

Figure 15: Attaching to a cdevDevice object

```
#include <cdevSystem.h>
#include <cdevDevice.h>

void main()
    {
    // ************************************************************
    // * Use the attachPtr and attachRef methods to attach to
    // * devices "MQB1S01" && "MQB1S02".
    // ************************************************************
    cdevDevice * devicePtr = cdevDevice::attachPtr("MQB1S01");
    cdevDevice & deviceRef = cdevDevice::attachRef("MQB1S02");

    // ************************************************************
    // * To use the long-hand approach, obtain a pointer to the
    // * default cdevSystem (or any other cdevSystem object).
    // ************************************************************
    cdevSystem & default = cdevSystem::defaultSystem();

    // ************************************************************
    // * Next, use the getDevice method of the system to obtain
    // * a pointer to the specific device (MQB1S03).
    // ************************************************************
    cdevDevice * dev = default.getDevice("MQB1S03");
    }
```

Getting and Setting Context for a cdevDevice

In CDEV, *context* refers to a collection of options that may be used to alter the way a CDEV device responds to a specific message. How context information effects the bahvior of a device is defined by the associated cdevService. Context may be used to set any number of I/O options, such as timeouts, data acquisition specifications, or flags to control optional return data. The context is stored and retrieved in the form of a cdevData object containing a collection of tagged data items (called *properties*).

The example below illustrates how the caller may obtain a copy of the context of a device, install a new context for a specific call, and then restore the previous context to the device.

Note: Whenever the context of a device is directly manipulated, each of its underlying cdevRequestObjects are updated to reflect the new change. Therefore, altering the context directly in the cdevDevice is not recommended. The preferred approach is to obtain a copy of the cdevRequestObject and alter the context there.

Figure 16: Altering the context of a cdevDevice object

```
#include <cdevSystem.h>
#include <cdevDevice.h>
#include <cdevData.h>

void main()
   {
   // ***********************************************************
   // * Obtain a pointer to the cdevDevice “MQB1S01”
   // ***********************************************************
   cdevDevice * devicePtr = cdevDevice::attachPtr(“MQB1S01”);
   cdevData oldContext, newContext;

   // ***********************************************************
   // * Indicate in the newContext cdevData object that devices
   // * should return their value, status and severity information
   // ***********************************************************
   newContext.insert(“value”, 1);
   newContext.insert(“status”, 1);
   newContext.insert(“severity”, 1);

   // ***********************************************************
   // * Preserve the contexts of the original device context, and
   // * then install the new device context.
   // ***********************************************************
   oldContext = devicePtr->getContext();
   devicePtr->setContext(newContext);

   // ***********************************************************
   // * After performing the context specific operations, restore
   // * the old context to the device.
   // ***********************************************************
   devicePtr->setContext(oldContext);
   }
```

Sending Messages to a Device Synchronously

In most CDEV applications, the caller will want to send a message to a device and then wait until a response is received. In order to do this, the *send* method of the cdevDevice object should be used. The *send* method receives three parameters; the message to be sent, a cdevData object containing data required by the device, and a cdevData object to receive the data returned by the device.

The following example demonstrates reading an attribute of a device and handling each class of error inline. A return code of *CDEV_SUCCESS* indicates that the message was successfully transmitted and processed.

Figure 17: Using the send method of a cdevDevice object

```
#include <cdevDevice.h>
#include <cdevData.h>

void main()
   {
   // ************************************************************
   // * Obtain a pointer to the cdevDevice "MQB1S01"
   // ************************************************************
   cdevDevice * device = cdevDevice::attachPtr("MQB1S01");
   cdevData input, output;
   double value = 0.0;

   // ************************************************************
   // * Use the send method to obtain the value from the device.
   // * Because there is no outbound data, the parameter is NULL.
   // ************************************************************
   switch(device->send("get bdl", NULL, &output))
      {
      // **** Message transmitted and processed successfuly. ****
      case CDEV_SUCCESS:
         output.get("value", &value);
         printf("Value of MQB1S01 is %f\n", value);
         break;

      // ****** Unknown device or device/message mismatch ******
      case CDEV_INVALIDOBJ:
         printf("Unknown device or device/message mismatch\n");
         break;

      // * Communications error between application and service *
      case CDEV_NOTCONNECTED:
      case CDEV_IOFAILED:
      case CDEV_TIMEOUT:
         printf("Communications error while sending\n");
         break;

      // ****************** Generic error *********************
      case CDEV_ERROR:
      case default:
         printf("Unable to send message\n");
         break;
      }

   // ************************************************************
   // * Increment the value, and use the "set bdl" command
   // ************************************************************
   value += 1.0;
   input.insert("value", value);
   device->send("set bdl", &input, NULL);
   }
```

Sending Messages to a Device Asynchronously

There are two methods provided by the cdevDevice object for sending messages asynchronously; *sendNoBlock* and *sendCallback*.

Using sendNoBlock

The *sendNoBlock* method is identical in interface to the *send* method, however, if it is unable to immediately transmit to and receive a response from the device, it will not wait. In order to determine if a *sendNoBlock* operation has completed successfully, the caller must poll the *cdevSystem* object, implement a grouping scheme using the *cdevGroup* object, or explicitly test for new data in the result data object.

Managing sendNoBlock Messages with the cdevSystem object

To manage asynchronous messages using the cdevSystem object, the caller should use the *pend* or *poll* method. These methods will submit the message to the underlying device, and will wait for a period of time for a response to arrive. If all messages have not been processed during the specified (or default) period of time, these methods will return a *CDEV_TIMEOUT* status code. If all transactions have been processed, these methods will return a *CDEV_SUCCESS* status code.

Grouping sendNoBlock Messages with cdevGroup

An alternative way to manage messages transmitted with sendNoBlock is to create and *start* a *cdevGroup* object prior to sending the first message. The application then transmits all of the messages that it wishes to manage within a single group. Once all messages have been transmitted, the caller will *end* the *cdevGroup* and then may call the *pend* member function of the *cdevGroup* object until the return value is *CDEV_SUCCESS*. The caller should always specify a discrete amount of time to wait when using the *pend* operation, otherwise, the application can enter an indefinite wait.

Note: Once a cdevGroup object has been started, it will remain open until its end method is executed ***OR*** *until the pend method of the cdevGroup object is executed. If a sendNoBlock is executed after the group has been ended, it will not be managed by that group.*

The example below illustrates how to transmit multiple messages to a device using the *sendNoBlock* method and the *cdevGroup* object.

Figure 18: Using sendNoBlock to communicate with a cdevDevice

```
#include <cdevGroup.h>
#include <cdevDevice.h>
#include <cdevData.h>

void main()
    {
    // ***********************************************************
    // * Obtain a pointer to the cdevDevice "MQB1S01" and "MQB1S02"
    // ***********************************************************
    cdevDevice * device1 = cdevDevice::attachPtr("MQB1S01");
    cdevDevice * device2 = cdevDevice::attachPtr("MQB1S02");
    cdevGroup    group;
    cdevData     output1, output2;

    // ***********************************************************
    // * Start the cdevGroup object to group all of the requests.
    // ***********************************************************
    group.start();
```

Figure 18: Using sendNoBlock to communicate with a cdevDevice (continued)

```
// ***********************************************************
// * Use the sendNoBlock command to transmit the messages to
// * the selected devices.  The "get VAL" message requires no
// * input, so the outbound cdevData object is NULL.
// ***********************************************************
device1->sendNoBlock("get VAL", NULL, &output1);
device2->sendNoBlock("get VAL", NULL, &output2);

// ***********************************************************
// * Use the end method to terminate the group.
// ***********************************************************
group.end();

// ***********************************************************
// * Use the pend method of the group to wait for 1 second
// * for all of the messages to be processed.  The allFinished
// * method can be used to obtain the completioon status of
// * the group (0=NOT DONE, 1=DONE).
// *
// * As a simple example, this function will wait no more than
// * 5 seconds for all messages to be processed.
// ***********************************************************
for(int i=0; i<5 && !group.allFinished(); i++)
    {
    group.pend(1.0);
    }
}
```

Using deferred groups for repeating lists of operations

A cdevGroup may be placed into a *deferred* mode prior to starting it, and in this case messages will not be transmitted until the group is flushed. After all operations within the group have completed, the group may be flushed again, causing the same set of operations to be executed again. See the reference manual for details.

Using sendCallback

The *sendCallback* method provides the second mechanism for transmitting messages asynchronously. This method is used in CDEV for a variety of purposes, the most common of these is to establish monitors on a specific cdevDevice attribute. Once the monitor has been established, the user specified callback function will be contacted whenever the property's value is altered.

This method is more complex than the others because it **REQUIRES** the caller to utilize a *cdevSystem* object in order to poll the underlying services, and it requires the caller to predefine a *cdevCallbackFunction* to be executed when the message has been processed.

The *cdevCallbackFunction* has a very specific prototype that the caller must comply with. The user specified callback function must be of type void, and must receive the following parameters.

1. **int status**: This is equivalent to the return status of a synchronous *send* call. It should receive one of the error codes that are defined in *cdevErrCode.h*.

2. **void * arg**: This is a pointer to data that was specified by the user when creating the *cdevCallback* object.

3. **cdevRequestObject & obj**: This is the *cdevRequestObject* that the system used to transmit the message to the device.

4. **cdevData & data**: This is the cdevData object that contains the output generated by the device when it processed the message.

The *cdevCallbackFunction* and a user provided argument are passed to the *sendCallback* method through the use of the *cdevCallback* object. The *cdevCallback* object is a simple container class.

Figure 19: Using the sendCallback method of a cdevDevice object

```
#include <cdevSystem.h>
#include <cdevDevice.h>
#incldue <cdevRequestObject.h>
#include <cdevCallback.h>
#include <cdevData.h>

// *****************************************************************
// * callback:
// * This is the callback function that will be called when the
// * message has been processed to completion.  This function is
// * disregarding CDEV_DISCONNECTED and CDEV_RECONNECTED messages.
// *****************************************************************
void callback(int status,
             void * arg,
             cdevRequestObject & req,
             cdevData & data)
   {
   double result;
   int & userFlag = *arg;

   // *************************************************************
   // * Save the completion status and print the result
   // *************************************************************
   if(status!=CDEV_DISCONNECTED && status!=CDEV_RECONNECTED)
      {
      userFlag = status;
      data.get(“value”, &result);
      printf(“I have received value %f”, result);
      }
   }

void main()
   {
   // *************************************************************
   // * Integer flag for detecting callback completion
   // *************************************************************
   int userFlag = 100;

   // *************************************************************
   // * Obtain a reference to the default system.
   // *************************************************************
   cdevSystem & default = cdevSystem::defaultSystem();

   // *************************************************************
   // * Obtain a pointer to the device “MQB1S01”.
   // *************************************************************
   cdevDevice * device = default.getDevice(“MQB1S01”);
```

Figure 19: Using the sendCallback method of a cdevDevice object (continued)

```
    // ***********************************************************
    // * Construct the cdevCallback object that will be used to
    // * specify the callback function and the user argument.
    // ***********************************************************
    cdevCallback cb(callback, &userFlag);

    // ***********************************************************
    // * Transmit the message to the device using the sendCallback
    // * mechanism.  Since the message requires no outbound data,
    // * the outbound cdevData object is NULL.
    // ***********************************************************
    if(device->sendCallback("get current", NULL, cb)==CDEV_SUCCESS)
        {
        // *******************************************************
        // * Now, poll the cdevSystem object until the message has
        // * been successfully processed.
        // *******************************************************
        while(userFlag==100) default.poll();
        }
    }
```

5. Using the *cdevRequestObject* Object

Overview of the cdevRequestObject Class

The *cdevRequestObject* C++ class is the provides the user with a higher performance interface to devices in CDEV. Each time a *cdevDevice* receives a message string, it must parse the message and then connect to the appropriate service. The *cdevRequestObject* allows the user to bind a *device* name to a *message* and then connect to the underlying service only ***once***. Thereafter, the *cdevRequestObject* will remain connected to the service and can process its associated message with much higher efficiency.

A *cdevRequestObject* may be created using the static member functions ***attachPtr*** or ***attachRef***. If one of these methods is used, the object will be created within the context of the default *cdevSystem*. If the application wishes to create a device within a *cdevSystem* other than the default, it must first obtain a pointer to the associated *cdevDevice* object from the desired system, and then use the ***getRequestObject*** method of that *cdevDevice*.

The message associated with a *cdevRequestObject* may be sent to a device using one of the three send member functions, these are: ***send, sendNoBlock****, and* ***sendCallback***. The syntax and functionality of these methods is described in the section below.

Input and output to the specified send methods is managed through the use of *cdevData* objects. Unlike the *cdevRequestObject* object which is bound to a specific system, the *cdevData* object is completely independent. These objects store the data values (called *properties*) that are sent to and received from devices. The application may indicate the *properties* that it wishes to receive as output from the *cdevRequestObject* object by specifying them in a special *cdevData* object called the *context*.

Examples at the end of this section will illustrate the correct ways to implement these operations.

Public Member Functions of the cdevRequestObject Class

attachRef *static cdevRequestObject& attachRef (char *device, char * msg);*

Obtains a reference to a *cdevRequestObject* object by specifying the name of the device and the message string. By default, the new object will be managed by the default *cdevSystem*.

attachRef *static cdevRequestObject& attachRef (cdevDevice &dev, char * msg);*

Obtains a reference to a *cdevRequestObject* object by providing a reference to the associated *cdevDevice* object and the message string. By default, the new object will be managed by the default *cdevSystem*.

attachPtr *static cdevRequestObject* attachPtr (char *device, char * msg);*

Obtains a pointer to a *cdevRequestObject* by specifying the name of the device and the message string. By default, the new object will be managed by the default *cdevSystem*.

attachPtr *static cdevRequestObject* attachPtr (cdevDevice &dev, char * msg);*

Obtains a pointer to a *cdevRequestObject* by providing a reference to the associated *cdevDevice* object and the message string. By default, the new object will be managed by the default *cdevSystem*.

detach *static void detach (cdevRequestObject& dev);*

Removes a referenced cdevRequestObject object from its associated cdevSystem object. Ordinary applications should never use this command.

detach *static void detach (cdevRequestObject* dev);*

Detaches the cdevRequestObject object specified by dev from its associated cdevSystem object. Ordinary applications should never use this command.

message *char *message (void) const;*

Retrieves the message string that is associated with this cdevRequestObject.

device *cdevDevice &device (void) const;*

Retrieves a reference to the cdevDevice object that is associated with this cdevRequestObject.

system *cdevSystem& system (void) const;*

Retrieves a reference to the underlying cdevSystem object that manages this cdevRequestObject.

service *cdevService& service (void) const;*

Retrieves a reference to the underlying cdevService object that this cdevRequestObject is attached to.

getState *int getState (void);*

Obtains the state of the underlying device. This function returns one of the following values as defined in *cdevErrCode.h*.

CDEV_STATE_CONNECTED:	Object is connected.
CDEV_STATE_NOTCONNECTED:	Object is not connected.
CDEV_STATE_INVALID:	Object is invalid.

The service developer is responsible for implementing this function correctly in the service related cdevRequestObject.

getAccess *int getAccess (void);*

Obtains access control information about the underlying device. This function returns one of the following values as defined in cdevErrCode.h.

CDEV_ACCESS_NONE:	No access to attribute.

CDEV_ACCESS_READONLY: Read-only access.

CDEV_ACCESS_WRITE: Read-write access.

The service developer is responsible for implementing this function correctly in the service related cdevRequestObject.

setContext

int setContext (cdevData& cxt);

Used to insert a cdevData object containing tagged values that control optional behavior of the underlying device. The context is often used to specify which properties (value, status, severity) a device returns in response to a "get" message. The service developer may override the default behavior of this method to better accomodate the requirements of the service.

getContext

cdevData & getContext (void);

Retrieves a reference to the cdevData object that contains the context for a specific cdevRequestObject. The service developer may override the default behavior of this method to better accomodate the requirements of the service.

getPrivate

*void * getPrivate (void);*

Retrieves a pointer to a data object that was placed in this object using the setPrivate function.

setPrivate

*void setPrivate (void * data);*

Associates a user specified data object with this *cdevRequestObject*. The pointer can be retrieved later using the *getPrivate* method.

send

int send (cdevData &out, cdevData& result);
*int send (cdevData *out, cdevData& result);*
int send (cdevData &out, cdevData result);*
*int send (cdevData *out, cdevData* result);*

The send function is the standard method for synchronously communicating with a device. The *out* cdevData object contains any property values that the device will need to perform the task. The *result* cdevData object will contain the output properties that resulted from the call. The service developer is responsible for implementing this method in the service specified cdevRequestObject. This function will return one of the error code defined in cdevErrCode.h.

Figure 20: Return codes generated by the send method

```
CDEV_SUCCESS:        The message was processed successfully.
CDEV_ERROR:          Failed to process message.
CDEV_INVALIDOBJ:     Invalid CDEV request object used.
CDEV_INVALIDARG:     Invalid argument passed to CDEV call.
CDEV_INVALIDSVC:     Wrong service during dynamic loading.
CDEV_NOTCONNECTED:   Not connected to low level network
                     service.
CDEV_IOFAILED:       Low level network service IO failed.
CDEV_CONFLICT:       Conflicts of data types or tags.
CDEV_NOTFOUND:       Cannot find specified data in cdevData.
CDEV_TIMEOUT:        Time out.
CDEV_CONVERT:        cdevData conversion error.
CDEV_OUTOFRANGE:     Value out of range for device attribute.
CDEV_NOACCESS:       Insufficient access to perform request.
CDEV_ACCESSCHANGED:  Change in access permission of device.
CDEV_DISCONNECTED:   The service has lost contact with the
                     device.
CDEV_RECONNECTED:    The service has regained contact with the
                     device.
```

sendNoBlock

int sendNoBlock (cdevData &out, cdevData &result);
*int sendNoBlock (cdevData *out, cdevData &result);*
*int sendNoBlock (cdevData &out, cdevData *result);*
*int sendNoBlock (cdevData *out, cdevData *result);*

The *sendNoBlock* method uses the same parameters and syntax as the *send* method. However, rather than waiting for the underlying service to respond to the request, this function will return immediately.The caller may use a cdevGroup object in order to detect when the transaction has been completed. The service developer is responsible for implementing this method in the service's cdevRequestObject.

sendCallback

int sendCallback (cdevData &out, cdevCallback &callback);
*int sendCallback (cdevData *out, cdevCallback &callback);*

The *sendCallback* function is the standard method for asynchronously communicating with a device. Rather than providing a *result* cdevData object, this method requires the user to provide the address of a cdevCallback object. This object contains a user supplied pointer and the address of a function to call when the message has been successfully processed. The service developer is responsible for implementing this method in the service specified cdevRequestObject.

Sample Code

The following sample applications illustrate many of the uses of the cdevRequestObject object. Additionally, these examples show in depth usage of the cdevData object and the concept of context.

Attaching to a cdevRequestObject Object

Within a cdevSystem object, a cdevRequestObject is created only once for any specified *device name* / *message string* combination. After that, all requests for that combination are given a pointer to the same cdevRequestObject object. This technique is called *attaching* to a cdevRequestObject.

There are two approaches to attaching to a *cdevRequestObject*; through a specific *cdevDevice* object, or through the *cdevRequestObject* interface. Attaching to the device using the *cdevDevice* object has the benefit of making it possible to manage the *cdevRequestObject* using the *cdevSystem* object associated with that device, rather than using the default system. Using the *cdevRequestObject* interface directly always places the new request object into the default system, but has the benefit of reducing the complexity of the source code by hiding the use of the *cdevDevice* and the *cdevSystem* objects.

The example below illustrates how to attach to a cdevRequestObject with a device named “MQB1S01” and the message “get VAL” using the cdevDevice interface.

Figure 21: Obtaining a cdevRequestObject from a cdevDevice object

```
#include <cdevSystem.h>
#include <cdevDevice.h>
#include <cdevRequestObject.h>

void main()
   {
   // ************************************************************
   // * LONG-HAND METHOD
   // * The first step in this approach is to obtain a pointer or
   // * reference to the desired system.  For this example, the
   // * defaultSystem will be used.
   // ************************************************************
   cdevSystem & default = cdevSystem::defaultSystem();

   // ************************************************************
   // * Next, use the getDevice method of the system to obtain
   // * a pointer to the specific device.
   // ************************************************************
   cdevDevice * device = default.getDevice(“MQB1S01”);

   // ************************************************************
   // * Finally, use the getRequestObject method of the cdevDevice
   // * class to obtain a cdevRequestObject for the message
   // * “get VAL” on device “MQB1S01”.
   // ************************************************************
   cdevRequestObject *req = device->getRequestObject(“get bdl”);

   // ************************************************************
   // * SHORT-HAND METHOD
   // * The following shortcut code directly creates a
   // * cdevRequestObject and installs it in the default
   // * cdevSystem object.
   // ************************************************************
   cdevRequestObject *req1 =
      cdevRequestObject::attachPtr(“MQB1S01”, “get bdl”);
   }
```

Getting and Setting Context for a cdevRequestObject

In CDEV, *context* refers to a collection of options that may be used to alter the way a CDEV device responds to a specific message. How context information effects the bahvior of a device is defined by the associated cdevService. Context may be used to set any number of I/O options, such as timeouts, data acquisition specifications, or flags to control optional return data. The context is stored and retrieved in the form of a cdevData object containing a collection of tagged data items (called *properties*).

The example below illustrates how the caller may obtain a copy of the context of a cdevRequestObject, install a new context for a specific call, and then restore the previous context to the object.

Figure 22: Altering the context of a cdevRequestObject object

```
#include <cdevSystem.h>
#include <cdevDevice.h>
#include <cdevRequestObject.h>
#include <cdevData.h>

void main()
   {
   cdevData oldContext, newContext;
   // **********************************************************
   // * Obtain a cdevRequestObject for “MQB1S01” / “get bdl”
   // **********************************************************
   cdevRequestObject *req =
      cdevRequestObject::attachPtr(“MQB1S01”, “get bdl”);

   // **********************************************************
   // * Indicate in the newContext cdevData object that this
   // * device should return its value, status and severity
   // **********************************************************
   newContext.insert(“value”, 1);
   newContext.insert(“status”, 1);
   newContext.insert(“severity”, 1);

   // **********************************************************
   // * Preserve the contexts of the original device context, and
   // * then install the new device context.
   // **********************************************************
   oldContext = req->getContext();
   req->setContext(newContext);

   // **********************************************************
   // * After performing the context specific operations, restore
   // * the old context to the device.
   // **********************************************************
   req->setContext(oldContext);
   }
```

Sending Messages to a Device Synchronously

In most CDEV applications, the caller will want to send a message to a device and then wait until a response is received. In order to do this, the *send* method of the cdevRequestObject object may be used. The *send* method receives two parameters; a cdevData object containing data required by the device, and a cdevData object to receive the data returned by the device.

The following example demonstrates how to read an attribute of a device, and handle each class of error inline. A return code of CDEV_SUCCESS indicates that the message was successfully transmitted and processed.

Figure 23: Using the send method of a cdevRequestObject object

```
#include <cdevRequestObject.h>
#include <cdevData.h>

void main()
   {
   // ***********************************************************
   // * Obtain a cdevRequestObject for "MQB1S01" / "get bdl"
   // ***********************************************************
   cdevRequestObject *req =
      cdevRequestObject::attachPtr("MQB1S01", "get bdl");
   cdevData input, output;
   double value = 0.0;

   // ***********************************************************
   // * Use the send method to obtain the value from the device.
   // * Since there is no outbound data, the parameter is NULL
   // ***********************************************************
   switch(req->send(NULL, &output))
      {
      // **** Message transmitted and processed successfuly. ****
      case CDEV_SUCCESS:
         output.get("value", &value);
         printf("Value of MQB1S01 is %f\n", value);
         break;

      // ******* Unknown device or device/message mismatch ******
      case CDEV_INVALIDOBJ:
         printf("Unknown device or device/message mismatch\n");
         break;

      // * Communications error between application and service *
      case CDEV_NOTCONNECTED:
      case CDEV_IOFAILED:
      case CDEV_TIMEOUT:
         printf("Communications error while sending\n");
         break;

      // ************ A generic error has occurred *************
      case CDEV_ERROR:
      case default:
         printf("Unable to send message\n");
         break;
      }
   }
```

Sending Messages to a Device Asynchronously

There are two methods provided by the cdevRequestObject object for sending messages asynchronously; *sendNoBlock* and *sendCallback*.

Using sendNoBlock

The *sendNoBlock* method is identical in interface to the *send* method, however, if it is unable to immediately transmit to and receive a response from the device, it will not wait. In order to determine if a *sendNoBlock* operation has completed successfully, the caller must poll the *cdevSystem* object, use the *cdevGroup* object, or explicitly test for new data in the result object.

Managing sendNoBlock Messages with the cdevSystem object

To manage asynchronous messages using the cdevSystem object, the caller should use the *pend* or *poll* method. These methods will submit the message to the underlying device, and will wait for a period of time for a response to arrive. If all messages have not been processed during the specified (or default) period of time, these methods will return a *CDEV_TIMEOUT* status code. If all transactions have been processed, these methods will return a *CDEV_SUCCESS* status code.

Grouping sendNoBlock Messages with cdevGroup

An alternative way to manage messages transmitted with *sendNoBlock*, is to create and *start* a *cdevGroup* object prior to sending the first message. The application will then transmit all of the messages that it wishes to manage within a single group. Once all messages have been transmitted, the caller will *end* the *cdevGroup* and then may call the *pend* member function of the *cdevGroup* object until the return value is *CDEV_SUCCESS*. The caller should always specify a discrete amount of time to wait when using the *pend* operation, otherwise, the application can enter an indefinite wait.

Note: Once a cdevGroup object has been started, it will remain open until its end method is executed ***OR*** *until the pend method of the cdevGroup object is executed. If a sendNoBlock is executed after the group has been ended, it will not be managed by that group.*

Figure 24: Using sendNoBlock to communicate with a cdevRequestObject

```
#include <cdevGroup.h>
#include <cdevRequestObject.h>
#include <cdevData.h>

void main()
   {
   cdevRequestObject *req1, *req2;
   cdevGroup          group;
   cdevData           output1, output2;
   // ***********************************************************
   // * Obtain cdevRequestObjects for “MQB1S01” and “MQB1S02”
   // ***********************************************************
   req1 = cdevRequestObject::attachPtr(“MQB1S01”, “get bdl”);
   req2 = cdevRequestObject::attachPtr(“MQB1S02”, “get bdl”);

   // ***********************************************************
   // * Start the cdevGroup object to group all of the requests.
   // ***********************************************************
   group.start();
```

Figure 24: Using sendNoBlock to communicate with a cdevRequestObject (continued)

```
// *************************************************************
// * Use the sendNoBlock command to transmit the messages to
// * the selected devices.  The "get bdl" message requires no
// * input, so the outbound cdevData object is NULL.
// *************************************************************
req1->sendNoBlock(NULL, &output1);
req2->sendNoBlock(NULL, &output2);

// *************************************************************
// * Use the end method to terminate the group.
// *************************************************************
group.end();

// *************************************************************
// * Use the pend method of the group to wait for 1 second
// * for all of the messages to be processed.  The allFinished
// * method can be used to obtain the completion status of
// * the group (0=NOT DONE, 1=DONE).
// *
// * As a simple example, this function will wait no more than
// * 5 seconds for all messages to be processed.
// *************************************************************
for(int i=0; i<5 && !group.allFinished(); i++)
    {
    group.pend(1.0);
    }
}
```

Using deferred groups for repeating lists of operations

A cdevGroup may be placed into a *deferred* mode prior to starting it, and in this case messages will not be transmitted until the group is flushed. After all operations within the group have completed, the group may be flushed again, causing the same set of operations to be executed again.

Using sendCallback

The *sendCallback* method provides the second mechanism for transmitting messages asynchronously. This method is used in CDEV for a variety of purposes, the most common of these is to establish monitors on a specific device property. Once the monitor has been established, the user specified callback function will be contacted whenever the property's value is altered.

This method is more complex than the others because it requires the caller to utilize a *cdevSystem* object in order to poll the underlying services, and it requires the caller to predefine a *cdevCallbackFunction* to be executed when the message has been processed.

The *cdevCallbackFunction* has a very specific prototype that the caller must comply with. The user specified callback function must be of type void, and must receive the following parameters.

1. **int status**: This is equivalent to the return status of a synchronous *send* call. It should receive one of the values as defined in *cdevErrCode.h*.

2. **void * arg**: This is a pointer to data that was specified by the user when creating the *cdevCallback* object.

3. **cdevRequestObject & obj**: This is the *cdevRequestObject* that the system used to transmit the message to the device.

4. **cdevData & data**: This is the cdevData object that contains the output generated by the device when it processed the message.

The *cdevCallbackFunction* and a user provided argument are passed to the *sendCallback* method through the use of the *cdevCallback* object. The *cdevCallback* object is a simple container class.

Figure 25: Using the sendCallback method of a cdevRequestObject object

```
#include <cdevSystem.h>
#include <cdevDevice.h>
#include <cdevRequestObject.h>
#include <cdevCallback.h>
#include <cdevData.h>

// ****************************************************************
// * callback:
// * This is the callback function that will be called when the
// * message has been processed to completion.  This function is
// * disregarding CDEV_DISCONNECTED and CDEV_RECONNECTED messages.
// ****************************************************************
void callback(int status,
             void * arg,
             cdevRequestObject & req,
             cdevData & data)
   {
   double result;
   int & userFlag = *arg;

   // ************************************************************
   // * Save the completion status and print the result
   // ************************************************************
   if(status!=CDEV_DISCONNECTED && status!=CDEV_RECONNECTED)
      {
      userFlag = status;
      data.get("value", &result);
      printf("I have received value %f", result);
      }
   }

void main()
   {
   // ************************************************************
   // * Integer flag for detecting callback completion
   // ************************************************************
   int userFlag = 100;

   // ************************************************************
   // * Obtain a reference to the default system.
   // ************************************************************
   cdevSystem & default = cdevSystem::defaultSystem();

   // ************************************************************
   // * Obtain a cdevRequestObject for "MQB1S01" / "get bdl".
   // ************************************************************
   cdevRequestObject * req =
      default.getDevice("MQB1S01")->getRequestObject("get bdl");
```

Figure 25: Using the sendCallback method of a cdevRequestObject object (continued)

```
// *************************************************************
// * Construct the cdevCallback object that will be used to
// * specify the callback function and the user argument.
// *************************************************************
cdevCallback cb(callback, &userFlag);

// *************************************************************
// * Transmit the message to the device using the sendCallback
// * mechanism.  Since the message requires no outbound data,
// * the outbound cdevData object is NULL.
// *************************************************************
if(req->sendCallback(NULL, cb)==CDEV_SUCCESS)
    {
    // *********************************************************
    // * Now, poll the cdevSystem object until the message has
    // * been successfully processed.
    // *********************************************************
    while(userFlag==100) default.poll();
    }
}
```

6. Using the *cdevData* Object

Overview of the cdevData Class

The cdevData C++ class is a self describing data object. This class is the primary mechanism for data interchange within the CDEV system. The cdevData object is capable of storing and retrieving tagged data items of all the primitive data types, as well as character strings and time stamps. These data items may be scalar or multi-dimensional arrays.

cdevData Tags

A tag is a unique 32 bit integer that may be defined by the CDEV system, individual services or by an application. Each tag integer has a corresponding character string identifier. By using the character string identifier to obtain the integer tag, applications and services are insulated from problems caused by internal tag renumbering.

XDR Packaging of cdevData Objects

In addition to storing data, the cdevData object can also encapsulate its contents into an XDR buffer for portable transport between platforms. The buffer can then be decoded and reassembled into a cdevData when it has been received. To ensure best performance, the XDR mechanism only transports the tag integer values, rather than the tag character string values. Therefore, client/server applications using the XDR transport must verify that they are using the same version of CDEV in order to avoid tag number mismatches.

Public Functions of the cdevData Class

tagC2I

*static int tagC2I (char *ctag, int *tag);*

Converts a character string tag name to its unique integer identifier. The function returns CDEV_SUCCESS if the conversion was successful, otherwise it returns CDEV_ERROR.

tagI2C

*static int tagI2C (int tag, char * &ctag);*

Converts a unique integer tag number to its related character string tag. The function returns CDEV_SUCCESS if the conversion was successful, otherwise it returns CDEV_ERROR.

insertTag

*static void insertTag(int tag, char *ctag);*

Adds a new, unique tag identifier to the static table of tags. Both the tag integer and the character string must be unique. Returns CDEV_SUCCESS on success, or CDEV_ERROR on failure.

operator =

cdevData & operator = (cdevData & data);

This is the assignment operator for the class. It will copy the exact contents of the cdevData object specified by data to the current cdevData object.

Cast operators

operator char (void);
operator short (void);
operator unsigned short (void);
operator int (void);
operator unsigned int (void);
operator long (void);
operator unsigned long (void);
operator float (void);
operator double (void);

Directly extracts a scalar value that is stored in the “value” data item, and returns it as the value of the current cdevData object.

asciiDump

*void asciiDump (FILE * fp);*

Outputs the complete contents of the cdevData object to a file. If no file pointer is specified, then stdout will be used.

xdrSize

*int xdrSize (size_t * bufLen, size_t * elementCount);*

Calculates the size of the buffer (*bufLen*) that will be required to store this cdevData object as represented by XDR. The total number of data items (*elementCount*) that will be placed in the buffer is also calculated. The values obtained from this function may be passed to the *xdrExport* method in order to use a pre-allocated data buffer.

xdrExport

*int xdrExport (char ** buf, size_t * bufLen);*

Allocates a buffer (*buf*) of sufficient size to hold the XDR representation of this cdevData object. It will then translate all data items stored within the object to XDR and write this data to the buffer. The size of the allocated buffer will be provided to the caller in the *bufLen* variable.

xdrExport

*int xdrExport (char * buf, size_t bufLen, size_t count);*

Populates the preallocated buffer (*buf*) with the XDR representation of the contents of this cdevData object. The buffer length (*bufLen*) and number of elements (*count*) must have been calculated in advance using the *xdrSize* method.

xdrImport

*int xdrImport (char * buf, size_t bufLen);*

Decodes the caller supplied buffer (*buf*) from XDR representation and populates the cdevData object with the contents. The caller must specify the number of bytes in the XDR buffer (*bufLen*).

remove

void remove(void);
void remove(int tag);
*void remove(char * ctag);*

Removes the data item specified by the unique tag name or integer from this cdevData object. If the tag is not specified, then all data items will be removed from the object.

changeTag

int changeTag(int oldTag, int newTag);
*int changeTag(int oldTag, char *c_newTag);*
*int changeTag(char *c_oldTag, int newTag);*
*int changeTag(char *c_oldTag, char *c_newTag);*

Retags a data item that is currently stored in the cdevData object from “oldTag” to “newTag”. By using this method the data can be renamed internally without having to perform a copy or removal. This function is overloaded to support any permutation of character string tag or integer tag identifier.

getType

cdevDataTypes getType(int tag);
*cdevDataTypes getType(char *ctag);*

Retrieves the enumerated data type of the specified tagged data item within this cdevData object. The following enumerated types are defined in the file *cdevTypes.h*.

CDEV_BYTE	8 bit unsigned character
CDEV_INT16	16 bit signed integer
CDEV_UINT16	16 bit unsigned integer
CDEV_INT32	32 bit signed integer
CDEV_UINT32	32 bit unsigned integer
CDEV_FLOAT	single precision floating point
CDEV_DOUBLE	double precision floating point
CDEV_STRING	NULL terminated character string
CDEV_TIMESTAMP	CDEV time stamp
CDEV_INVALID	invalid or unknown data type

getDim

*int getDim(int tag, size_t *dim);*
*int getDim(char *ctag, size_t *dim);*

Retrieves the number of dimensions in a data item that is an array. If dim is 0, then the data item is scalar, otherwise, the data item is a “dim” dimensional array. This function is overloaded to support either the integer tag or character string tag.

getElems

*int getElems(int tag, size_t *elems);*
*int getElems(char *ctag, size_t *elems);*

Retrieves the number of data elements in all dimensions of a multi-dimensional array. If elems is 1, then the data item is scalar, otherwise, the data item is an array of “elems” elements. This function is overloaded to support either the integer tag or character string tag.

getBounds

int getBounds(int tag,
*cdevBounds * bounds,*
size_t numBounds);
*int getBounds(char * ctag,*
*cdevBounds * bounds,*
size_t numBounds);

Obtains the bounding dimensions of the array specified by the caller provided tag. The cdevBounds structure contains two integers: offset and length. The length variable represents the number of data elements in that dimension of the array. The offset integer is provided for use in specifying the position of arrays that are sub-sets of the actual data. A version of this function is available that uses an array of integers rather than the cdevBounds structure. This function is overloaded to support either the integer tag or character string tag.

setBounds

int setBounds(int tag,
*cdevBounds * bounds,*
size_t numBounds);
*int setBounds(char * ctag,*
*cdevBounds * bounds,*
size_t numBounds);

Specifies the bounding dimensions of a multi-dimensional array of data. In order to use this function, the number of dimensions in the array must be specified as a parameter to the insert command when placing the data into the cdevData object. The cdevBounds structure contains two integers: offset and length. The length variable represents the number of data elements in that dimension of the array. The offset integer is provided for use in specifying the position of arrays that are sub-sets of the actual data. A version of this function is available that uses an array of integers rather than the cdevBounds structure. This function is overloaded to support either the integer tag or character string tag.

insert (*scalar*)

int insert (int tag, BYTE data);
int insert (int tag, short data);
int insert (int tag, unsigned short data);
int insert (int tag, int data);
int insert (int tag, unsigned int data);
int insert (int tag, long data);
int insert (int tag, unsigned long data);
int insert (int tag, float data);
int insert (int tag, double data);
int insert (int tag, cdev_TS_STAMP data);

These methods allow the user to insert a single data item into the cdevData object using the specified tag. If another data item already occupies that tag within the cdevData object, the new data item will override the old one. Although only the integer tag prototypes are shown, these functions are overloaded to support both the integer and character string data tags.

insert (array)

*int insert (int tag, BYTE * data, size_t len, size_t ndim);*
*int insert (int tag, short * data, size_t len, size_t ndim);*
*int insert (int tag, unsigned short * data, size_t len, size_t ndim);*
*int insert (int tag, int * data, size_t len, size_t ndim);*
*int insert (int tag, unsigned int * data, size_t len, size_t ndim);*
*int insert (int tag, long * data, size_t len, size_t ndim);*
*int insert (int tag, unsigned long * data, size_t len, size_t ndim);*
*int insert (int tag, float * data, size_t len, size_t ndim);*
*int insert (int tag, double * data, size_t len, size_t ndim);*

These methods allow the user to insert array data items into the cdevData object using the specified tag. The len parameter is used to specify the total number of items in the array. The ndim parameter is used to specify the number of dimensions in the array (1 by default). Although only the integer tag prototypes are shown, these functions are overloaded to support both the integer and character string data tags.

insert (character string)

*int insert (int tag, char * data);*
*int insert (int tag, char ** data, size_t len, size_t ndim);*

These methods are used to add NULL terminated character strings to the cdevData object. These methods differ from the others because a single character string is treated as a scalar value. The len parameter (where used) specifies the total number of character strings in the array. The ndim parameter is used to specify the number of dimensions in the array (1 by default). Although only the integer tag prototypes are shown, these functions are overloaded to support both the integer and character string data tags.

get

*int get(int tag, BYTE * data);*
*int get(int tag, short * data);*
*int get(int tag, unsigned short * data);*
*int get(int tag, int * data);*
*int get(int tag, unsigned int * data);*
*int get(int tag, long * data);*
*int get(int tag, unsigned long * data);*
*int get(int tag, float * data);*
*int get(int tag, double * data);*
*int get(int tag, cdev_TS_STAMP * data);*

These methods are used to retrieve scalar values and arrays that are stored in the cdevData object. When retrieving an array, it is the caller's responsibility to call the *getElems* method to obtain the number of elements and then pre-allocate the buffer to receive the data. If the receiving data type does not match the data type stored in the cdevData object, then the data will be converted to the new type when it is extracted. Although only the integer tag prototypes are shown, these functions are overloaded to support both the integer and character string data tags.

get (character string)

*int get(int tag, char * data, size_t len);*
*int get(int tag, char ** data);*

These methods are used to retrieve character strings and arrays of character strings from a cdevData object. When obtaining a single character string, the length of the allocated buffer must be specified in the *len* parameter. When obtaining an array of character strings, the caller must pre-allocate the POINTERS for each element in the array. The cdevData object will allocate sufficient memory to each of the pointers to hold the individual character strings. It then becomes the responsibility of the caller to free the memory assigned to those pointers. If the receiving data type does not match the data type stored in the cdevData object, then the data will be converted to the new type when it is extracted. Although only the integer tag prototypes are shown, these functions are overloaded to support both the integer and character string data tags.

find

int find(int tag, void &data);*
*int find(char * ctag, void* &data);*

Retrieves a pointer to the data within the cdevData object. The caller is responsible for ensuring that the receiving pointer is of the correct data type for the data within the object. This function is overloaded to support both integer and character string tag identifiers.

Sample Code

The following sample program shows the steps necessary to add a new tag character string and tag integer to the global CDEV tag table. This method is most frequently used by the CDEV system and CDEV services.

Figure 26: Using insertTag to add a new tag

```
#include <cdevData.h>

void main()
   {
   // *****************************************************
   // * Unique tag number and character string
   // *****************************************************
   int    tag     = 100;
   char * tagName = "testTag";

   // *****************************************************
   // * Insert the new tag
   // *****************************************************
   cdevData::insertTag(tag, tagName);
   printf("Inserted tag %s as tag number %i\n", tagName, ta
   }
```

The following sample application illustrates the steps necessary to convert between character string and integer tag values using the *tagI2C* and *tagC2I* methods.

Figure 27: Using tagC2I and tagI2C to determine cdevData tag identifiers

```
#include <cdevData.h>

void main()
   {
   int    tag;
   char * tagName;
   // ******************************************************
   // * Insert a new tag into cdevData to illustrate this f
   // ******************************************************
   cdevData::insertTag(100, "testTag");

   // ******************************************************
   // * Now use the command tagI2C to obtain the character
   // * name of the tag from the tag number.
   // ******************************************************
   cdevData::tagI2C(100, tagName);

   // ******************************************************
   // * Using the newly obtained character string, call tag
   // * obtain the integer tag value.
   // ******************************************************
   cdevData::tagC2I(tagName, &tag);

   printf("Tag name %s is tag number %i", tagName, tag);
   }
```

The following sample application shows the steps necessary to place scalar data into a cdevData object using the *insert* method and then retrieve the data using the *get* method.

Figure 28: Inserting and retrieving scalar data items using a cdevData object

```
#include <cdevData.h>

void main()
   {
   cdevData data;
   int      x;
   double   y;

   // ************************************************************
   // Insert a short integer as the "value" property.
   // ************************************************************
   data.insert("value", (short int)1);

   // ************************************************************
   // * Insert a double as the "status" property
   // ************************************************************
```

Figure 28: Inserting and retrieving scalar data items using a cdevData object (continued)

```
data.insert("status", (double)2.0);

// ***********************************************************
// * Get the short stored in the "value" property as a double
// ***********************************************************
data.get("value", &y);
printf("value is %f\n", y);

// ***********************************************************
// * Get the double stored in the "status" property as an int
// ***********************************************************
data.get("status", &x);
printf("status is %i\n", x);
}
```

The following example illustrates the proper usage of the changeTag method to alter the tag identifier of a data item stored in the cdevData object.

Figure 29: Using the changeTag method of the cdevData object

```
#include <cdevData.h>

void main()
    {
    cdevData  data;
    short int x;

    // ***********************************************************
    // Insert a short integer as the "value" property.
    // ***********************************************************
    data.insert("value", (short int)1);

    // ***********************************************************
    // * Use the changeTag method to convert the tag identifier
    // * for the data item from "value" to "status".
    // ***********************************************************
    data.changeTag("value", "status");

    // ***********************************************************
    // * Use the get method to retrieve the short integer from the
    // * "status" property.
    // ***********************************************************
    data.get("status", &x);
    printf("Retrieved %i from the status property\n", x);
    }
```

The following sample application illustrates how to use the cdevData object to insert and retrieve a 1 dimensional array of doubles.

Figure 30: Using one-dimensional arrays with cdevData objects

```
#include <cdevData.h>

void main()
   {
   cdevData  data;
   size_t    count;
   double    inArray[5] = {1.0, 2.0, 3.0, 4.0, 5.0};
   double   *outArray;

   // ************************************************************
   // * Use the insert method to add the array to the value
   // * property of the cdevData object.
   // ************************************************************
   data.insert("value", inArray, 5);

   // ************************************************************
   // * Now we will extract the data that we just inserted.
   // * First, use the getElems method to determine the number of
   // * elements in the array.
   // ************************************************************
   data.getElems("value", &count);

   // ************************************************************
   // * Next allocate an array of doubles sufficient to hold
   // * the array stored in the cdevData object.
   // ************************************************************
   outArray = new double[count];

   // ************************************************************
   // * Finally, retrieve the data from the cdevData object using
   // * the get method.
   // ************************************************************
   data.get("value", outArray);

   // ************************************************************
   // * Output the result and then free any locally allocated
   // * memory.
   // ************************************************************
   for (int i=0; i<count; i++)
      {
      printf("Element %i is %f\n", i, outArray[i]);
      }
   delete outArray;
   }
```

The following example illustrates the correct method for inserting and retrieving a multi-dimensional array of doubles. This sample program also demonstrates the use of the getBounds and setBounds function calls, and how they are used to define the bounding dimensions for a multi-dimensional array.

Figure 31: Using multi-dimensional arrays with cdevData objects

```
#include <cdevData.h>

void main()
   {
   cdevData   data;
   int        i;
   size_t     nElems=1;
   size_t     nDim  =1;
   cdevBounds inbounds[2];
   double     inarray[2][5] =
   {
   {0.0, 1.0, 2.0, 3.0, 4.0},
   {5.0, 6.0, 7.0, 8.0, 9.0}
   };

   double     * outarray;
   cdevBounds * outbounds;

   // ############################################################
   // # Inserting multi-dimensional data into a cdevData object.
   // ############################################################

   // ************************************************************
   // * Set the length of each dimension of the array within the
   // * cdevBounds structure.  Also, set the offset to 0 for each
   // * dimension to specify that this array begins at the origen;
   // ************************************************************

   inbounds[0].offset = 0;// Offset of first dimension
   inbounds[0].length = 2;// Length of first dimension

   inbounds[1].offset = 0;// Offset of second dimension
   inbounds[1].length = 5;// Length of second dimension

   // ************************************************************
   // * Use the insert method to add the array to the value
   // * property of the cdevData object.  The third parameter
   // * specifies the exact number of elements that will be added,
   // * while the fourth parameter specifies that this will be a
   // * two dimensional array.
   // ************************************************************
   data.insert("value", (double *)inarray, 10, 2);

   // ************************************************************
   // * Use the setBounds method to specify the bounding
   // * dimensions of the multi-dimensional array.
   // ************************************************************
   data.setBounds("value", inbounds, 2);
```

Figure 31: Using multi-dimensional arrays with cdevData objects (continued)

```
// ##############################################################
// # Retrieving multi-dimensional data from a cdevData object.
// ##############################################################

// **************************************************************
// * Call the getDim method to obtain the number of dimensions
// * in the array.
// **************************************************************
data.getDim("value", &nDim);

// **************************************************************
// * Allocate a cdevBounds structure of sufficient size to
// * hold all dimensions of the array.
// **************************************************************
outbounds = new cdevBounds[nDim];

// **************************************************************
// * Use the getBounds methods to obtain the bounding
// * dimensions of the array.
// **************************************************************
data.getBounds("value", outbounds, 2);

// **************************************************************
// * Calculate the number of elements in the complete array.
// * This step is shown for illustrative purposes only.  The
// * result of this operation should be the same value that
// * would be returned by the getElems method.
// **************************************************************
for(i=0; i<nDim; i++) nElems *= outbounds[i].length;

// **************************************************************
// * Allocate an outbound buffer of sufficient size to hold
// * the data in all dimensions of the array.
// **************************************************************
outarray = new double[nElems];

// **************************************************************
// * Use the get method to obtain the data from the cdevData
// * object.
// **************************************************************
data.get("value", outarray);

// **************************************************************
// * Output the data retrieved.
// **************************************************************
for(i=0; i<nElems; i++) printf("%f\n", outarray[i]);

delete(outbounds);
delete(outarray);
}
```

The following example shows the correct method to insert and retrieve a character string using a cdevData object. The treatment of character strings in CDEV differs from ordinary values, because a character string is an array that is treated as a scalar. Therefore, the structure of the get method for a character string is unique.

Figure 32: Using character strings with cdevData objects

```
#include <cdevData.h>

void main()
    {
    cdevData data;
    char * inString = "This is my test string";
    char   outString[50];

    // ************************************************************
    // * Use the insert method to insert the character string into
    // * the cdevData object.
    // ************************************************************
    data.insert("value", inString);

    // ************************************************************
    // * Next, use the get function to retrieve the string.  The
    // * maximum length of the string to be retrieved is specified
    // * by the third parameter (50).
    // ************************************************************
    data.get("value", outString, 50);

    printf("String is: %s\n", outString);
    }
```

The next example illustrates the correct method for retrieving an array of character strings. This is a very special case because, with normal arrays, the caller must pre-allocate the array that the data will be read into when the get method is caller. With an array of character strings, the user is responsible for allocating only the array of POINTERS that each string will be assigned to. The get method will then allocate sufficient memory to each pointer to store the data.

When finished using the array, the user is responsible for freeing the memory assigned to each individual POINTER, and then freeing the array of pointers.

Figure 33: Using character string arrays with cdevData objects

```
#include <cdevData.h>

void main()
    {
    cdevData data;
    char **  outString;
    size_t   nElems;
    int      i;
```

Figure 33: Using character string arrays with cdevData objects (continued)

```
char *   inStrings[10] =
    {
    "string 0",
    "string 1",
    "string 2",
    "string 3",
    "string 4",
    "string 5",
    "string 6",
    "string 7",
    "string 8",
    "string 9"
    };

// ############################################################
// # Inserting character string arrays
// ############################################################

// ************************************************************
// * Place the array of character strings into the cdevData
// * object using the insert command. (by default, ndim=1)
// ************************************************************
data.insert("value", inStrings, 10);

// ############################################################
// # Retrieving character string arrays
// ############################################################

// ************************************************************
// * Use the getElems function to determine the total number
// * of STRINGS that will be retrieved.
// ************************************************************
data.getElems("value", &nElems);

// ************************************************************
// * Allocate an array of pointers, 1 for each element.
// ************************************************************
outString = new char *[nElems];

// ************************************************************
// * Use the get method to retrive the data from cdevData.
// ************************************************************
data.get("value", outString);

// ************************************************************
// * Output the data recieved from the cdevData object.
// ************************************************************
for(i=0; i<nElems; i++)
    printf("String %i: %s\n", i, outString[i]);
```

Figure 33: Using character string arrays with cdevData objects (continued)

```
    // ************************************************************
    // * Free the memory allocated by cdevData.
    // ************************************************************
    for(i=0; i<nElems; i++)
       if(outString[i]!=NULL) delete outString[i];

    // ************************************************************
    // * Free the locally allocated array of pointers.
    // ************************************************************
    delete outString;
    }
```

The next sample application shows how to use the find method of the cdevData object. The find method can be used to obtain a memory pointer to the data array within the cdevData object without allocating any new memory. However, because the find method does not perform any data type conversions, the caller is responsible for ensuring that a pointer to the correct data type is used.

This example shows how to use the find method as an alternative approach for obtaining an array of character strings.

Figure 34: Using the find method of the cdevData object

```
#include <cdevData.h>

void main()
    {
    char * inStrings[10] =
       {
       "string 0",
       "string 1",
       "string 2",
       "string 3",
       "string 4",
       "string 5",
       "string 6",
       "string 7",
       "string 8",
       "string 9"
       };
    cdevData      data;
    cdevDataTypes datatype;
    char **       outString;
    size_t        nElems;
    int           i;

    // ************************************************************
    // * Place the array of character strings into the cdevData
    // * object using the insert command.
    // ************************************************************
    data.insert("value", inStrings, 10);
```

Figure 34: Using the find method of the cdevData object (continued)

```
// ##############################################################
// # Retrieving a pointer to the data using the find method
// ##############################################################

// **************************************************************
// * For this example, we are going to ensure that the value
// * we are obtaining is an array of strings.
// * The getType method will be caller to ensure that the
// * data type of the value is CDEV_STRING, and then the
// * getElems method will be called to make sure that their
// * is more than one string in the cdevData object.
// **************************************************************
datatype = data.getType("value");
data.getElems("value", &nElems);

if(datatype==CDEV_STRING && nElems>0)
    {
    // ********************************************************
    // * Call find to obtain a copy of the data.
    // ********************************************************
    data.find("value", (void *&)outString);

    // ********************************************************
    // * Output the data recieved from the cdevData object.
    // ********************************************************
    while(outString[i]!=NULL)
        printf("String %i: %s\n", i++, outString[i]);
    }
}
```

7. Using the *cdevDirectory* Device

cdevDirectory Device

The cdevDirectory device is a cdevDevice object that provides an interface to directory services within CDEV. Directory services allow an application to obtain data that is specified in the device definition file (DDL file). Information in the DDL file identifies *services*, *devices* and *messages* that are supported by CDEV.

Attaching to the cdevDirectory

The cdevDirectory device can be accessed in the same manner as any other CDEV device. Their are two methods for obtaining a cdevDirectory device. The first approach is to get a pointer or reference to a cdevDevice object with the device name set to cdevDirectory: *“**cdevDevice::attachPtr(“cdevDirectory”)**”.*

The second approach allows the caller to obtain a pointer to the directory object from the system object.

The example below illustrates using the system object to obtain a pointer to a cdevDirectory device.

Figure 35: Attaching to the cdevDirectory device

```
// ****************************************************************
// * Include required CDEV header files
// ****************************************************************
#include <cdevSystem.h>
#include <cdevDevice.h>

void main()
    {
    // ***********************************************************
    // * Obtain a reference to the default cdevSystem object
    // ***********************************************************
    cdevSystem & system = cdevSystem::defaultSystem();

    // ***********************************************************
    // * Obtain a pointer to the “cdevDirectory” device using the
    // * getDevice member function of the default cdevSystem
    // * object.
    // ***********************************************************
    cdevDevice * directory = system.getDevice(“cdevDirectory”);
    .
    .
    .
    }
```

Messages Supported by cdevDirectory

An application communicates with the cdevDirectory device through the use of predefined message strings. The following messages are supported by the cdevDirectory device.

1. ***query:*** Identify the *devices* that are members of a DDL *class*.
2. ***queryClass:*** Identify the DDL *class* from which a *device* is instantiated.

3. ***queryAttributes:*** Identify all *attributes* supported by a *device* or a DDL *class.*
4. ***queryMessages:*** Identify all *messages* supported by a *device* or DDL *class.*
5. ***queryVerbs:*** Identify all *verbs* supported by a *device* or DDL *class.*
6. ***service:*** Identify the *service* that is used by a *device/message* pair.
7. ***serviceData:*** Identify *service* data specified for a *device/message* pair.
8. ***update:*** Add information to the *cdevDirectory* data structure.
9. ***validate:*** Verify that a *device* or DDL *class* contains certain properties.

"query"

The "query" message is submitted to the cdevDirectory object in order to obtain a list of matching *devices* that are instantiated from a specified DDL *class*. The class name is specified as an absolute string. The device name may be specified either as an absolute string or as a *regular expression*.

The request is submitted to the cdevDirectory device through its *send*, *sendCallback*, or *sendNoBlock* interface. The following table shows the inputs that must be placed in the inbound cdevData object prior to submitting the command, and the outputs that can be retrieved when the command is completed.

Figure 36: cdevData Input/Output associated with a "query" message

	Tag Name	*Data Type*	*Content*
Message Input	***device***	character string	Regular expression identifying the device names that should be retrieved.
	class	character string	Name of the DDL class that the devices should be instantiated from.
Message Output	***value***	array of strings	Names of matching devices

The following figure shows the source code necessary to request the names of all *devices* that are members of the DDL *class* "magnet".

Figure 37: Using the "query" Message with a cdevDirectory device

```
#include <cdevSystem.h>
#include <cdevDevice.h>
#include <cdevData.h>

void main()
    {
    // ***********************************************************
    // * Use the shorthand method to obtain a pointer to the
    // * cdevDirectory device within the default system.
    // ***********************************************************
    cdevDevice *dir = cdevDevice::attachPtr("cdevDirectory");
    cdevData input, output;

    // ***********************************************************
    // * Insert the name of the DDL class and the regular
    // * expression for the device.
    // ***********************************************************
    input.insert("class", "magnet");
    input.insert("device", ".*");

    // ***********************************************************
    // * Submit the request to the cdevDirectory object using the
    // * send command.
    // ***********************************************************
    if(dir->send("query", input, output)==CDEV_SUCCESS)
        {
        // *******************************************************
        // * If the command was successful, obtain a pointer to the
        // * list of names and output them on stdout.
        // *******************************************************
        char ** ptr;
        output.find("value", ptr);
        while(*ptr!=NULL) fprintf(stdout, "%s\n", ptr++);
        }
    }
```

"queryClass"

The "queryClass" message is submitted to the cdevDirectory object in order to identify the name of the DDL *class* that a *device* is instantiated from.

The request is submitted to the cdevDirectory device through its send, sendCallback, or sendNoBlock interface. The following table shows the inputs that must be placed in the inbound cdevData object prior to submitting the command, and the outputs that can be retrieved when the command is completed.

Figure 38: cdevData input/output associated with a "queryClass" message

	Tag Name	*Data Type*	*Content*
Message Input	***device***	character string	Name of the device to be queried.
Message Output	***value***	character string	Names of the associated class

The following figure shows the source code necessary to determine the name of the DDL class from which the device “MQB1S01” is derived.

Figure 39: Using the “queryClass” message with a cdevDirectory device

```
#include <cdevSystem.h>
#include <cdevDevice.h>
#include <cdevData.h>

void main()
    {
    cdevDevice *dir = cdevDevice::attachPtr(“cdevDirectory”);
    cdevData input, output;

    // **********************************************************
    // * Insert the name of the DDL class.
    // **********************************************************
    input.insert(“device”, “MQB1S01”);

    // **********************************************************
    // * Submit the request to the cdevDirectory object.
    // **********************************************************
    if(dir->send(“queryClass”, input, output)==CDEV_SUCCESS)
        {
        // *****************************************************
        // * If successful, output the name of the DDL class.
        // *****************************************************
        char * ptr;
        output.find(“value”, ptr);
        fprintf(stdout, “MQB1S01 is a %s\n”, value);
        }
    }
```

“queryAttributes”

The “queryAttributes” message is submitted to the cdevDirectory object in order to obtain a list of all *attributes* that are contained within a specified *device* or DDL *class*.

The request is submitted to the cdevDirectory device through its *send*, *sendCallback*, or *sendNoBlock* interface. The following table shows the inputs that must be placed in the inbound cdevData object prior to submitting the command, and the outputs that can be retrieved when the command is completed.

Figure 40: cdevData input/output associated with a “queryAttributes” message

	Tag Name	*Data Type*	*Content*
Message Input	***device*** or	character string	Name of the device to be queried.
	class	character string	Name of the DDL class to be queried.
Message Output	***value***	array of strings	Names of matching attributes

The following figure shows the source code necessary to request the names of all *attributes* that are defined within the DDL *class* “magnet”.

Figure 41: Using the "queryAttributes" message with a cdevDirectory device

```
#include <cdevSystem.h>
#include <cdevDevice.h>
#include <cdevData.h>

void main()
    {
    cdevDevice *dir = cdevDevice::attachPtr("cdevDirectory");
    cdevData input, output;

    // **********************************************************
    // * Insert the name of the DDL class.
    // **********************************************************
    input.insert("class", "magnet");

    // **********************************************************
    // * Submit the request to the cdevDirectory object using the
    // * send command.
    // **********************************************************
    if(dir->send("queryAttributes", input, output)==CDEV_SUCCESS)
        {
        // *****************************************************
        // * If the command was successful, obtain a pointer to
        // * the list of attributes and output them on stdout.
        // *****************************************************
        char ** ptr;
        output.find("value", ptr);
        while(*ptr!=NULL) fprintf(stdout, "%s\n", ptr++);
        }
    }
```

"queryMessages"

The "queryMessages" message is submitted to the cdevDirectory object in order to obtain a list of all *messages* that are contained within a specified *device* or DDL *class*.

The request is submitted to the cdevDirectory device through its *send*, *sendCallback*, or *sendNoBlock* interface. The following table shows the inputs that must be placed in the inbound cdevData object prior to submitting the command, and the outputs that can be retrieved when the command is completed.

Figure 42: cdevData input/output associated with a "queryMessages" message

	Tag Name	*Data Type*	*Content*
Message Input	***device*** or	character string	Name of the device to be queried.
	class	character string	Name of the DDL class to be queried.
Message Output	***value***	array of strings	Names of matching messages

The following figure shows the source code necessary to request the names of all *messages* that are defined within the DDL *class* “magnet”.

Figure 43: Using the “queryMessages” message with a cdevDirectory device

```
#include <cdevSystem.h>
#include <cdevDevice.h>
#include <cdevData.h>

void main()
    {
    cdevDevice *dir = cdevDevice::attachPtr(“cdevDirectory”);
    cdevData input, output;

    // ***********************************************************
    // * Insert the name of the DDL class.
    // ***********************************************************
    input.insert(“class”, “magnet”);

    // ***********************************************************
    // * Submit the request to the cdevDirectory object using the
    // * send command, and output the result if successful.
    // ***********************************************************
    if(dir->send(“queryMessages”, input, output)==CDEV_SUCCESS)
        {
        char ** ptr;
        output.find(“value”, ptr);
        while(*ptr!=NULL) fprintf(stdout, “%s\n”, ptr++);
        }
    }
```

“queryVerbs”

The “queryVerbs” message is submitted to the cdevDirectory object in order to obtain a list of all *verbs* that are contained within a specified *device* or DDL *class*.

The request is submitted to the cdevDirectory device through its *send*, *sendCallback*, or *sendNoBlock* interface. The following table shows the inputs that must be placed in the inbound cdevData object prior to submitting the command, and the outputs that can be retrieved when the command is completed.

Figure 44: cdevData input/output associated with a “queryVerbs” message

	Tag Name	*Data Type*	*Content*
Message Input	***device*** or	character string	Name of the device to be queried.
	class	character string	Name of the DDL class to be queried.
Message Output	***value***	character strings	Names of matching verbs.

The following figure shows the source code necessary to request the names of all *verbs* that are defined within the DDL *class* “magnet”.

Figure 45: Using the "queryVerbs" message with a cdevDirectory device

```
#include <cdevSystem.h>
#include <cdevDevice.h>
#include <cdevData.h>

void main()
   {
   cdevDevice *dir = cdevDevice::attachPtr("cdevDirectory");
   cdevData input;
   cdevData output;

   // ***********************************************************
   // * Insert the name of the DDL class.
   // ***********************************************************
   input.insert("class", "magnet");

   // ***********************************************************
   // * Submit the request to the cdevDirectory object using the
   // * send command, and output the result if successful.
   // ***********************************************************
   if(dir->send("queryVerbs", input, output)==CDEV_SUCCESS)
      {
      char ** ptr;
      output.find("value", ptr);
      while(*ptr!=NULL) fprintf(stdout, "%s\n", ptr++);
      }
   }
```

"service"

The "service" message is submitted to the cdevDirectory object in order to identify which CDEV *service* will respond to a specified *device* / *message* combination. The caller specifies the *device* name and the *message* string in the *input* cdevData object and can retrieve the result from the value entry in the resulting *output* cdevData object. This message will normally only be used internally by CDEV.

The request is submitted to the cdevDirectory device through its *send*, *sendCallback*, or *sendNoBlock* interface. The following table shows the inputs that must be placed in the inbound cdevData object prior to submitting the command, and the outputs that can be retrieved when the command is completed.

Figure 46: cdevData input/output associated with a "service" message

	Tag Name	*Data Type*	*Content*
Message Input	***device***	character string	Name of the CDEV device.
	message	character string	Name of the message.
Message Output	***value***	character string	Name of the service that will respond to the specified device and message.

The following figure shows the source code necessary to resolve the name of the service that will respond to the device "DEV1" and the message "get VAL".

Figure 47: Using the “service” message with a cdevDirectory device

```
#include <cdevSystem.h>
#include <cdevDevice.h>
#include <cdevData.h>

void main()
    {
    cdevDevice *dir = cdevDevice::attachPtr("cdevDirectory");
    cdevData input, output;

    // ************************************************************
    // * Insert the name of the device and message to be resolved.
    // ************************************************************
    input.insert("device", "DEV1");
    input.insert("message", "get VAL");

    // ************************************************************
    // Submit the request to the cdevDirectory object using the
    // send command, and output the result if successful.
    // ************************************************************
    if(dir->send("service", input, output)==CDEV_SUCCESS)
        {
        char * ptr;
        output.find("value", ptr);
        fprintf(stdout, "Service name is:%s\n", ptr);
        }
    }
```

“serviceData”

The “serviceData” message is submitted to the cdevDirectory object in order to obtain *service* specific data that is associated with a *device* or a *message*. The information will be returned to the caller in the output cdevData object in the form of tagged data items. This message will normally only be used in the development of a new CDEV service.

The request is submitted to the cdevDirectory device through its *send*, *sendCallback*, or *sendNoBlock* interface. The following table shows the inputs that must be placed in the inbound cdevData object prior to submitting the command, and the outputs that can be retrieved when the command is completed.

Figure 48: cdevData input/output associated with a “serviceData” message

	Tag Name	*Data Type*	*Content*
Message Input	***device***	character string	Name of the CDEV device.
	message	character string	Name of the message.
Message Output	***(service specific)***	character string	Data items specified in the service data section for the specified device / message combination.

The following figure shows the source code necessary to obtain the serviceData associated with the device “DEV1” and the message “get VAL”.

Figure 49: Using the "serviceData" message with a cdevDirectory device

```
#include <cdevSystem.h>
#include <cdevDevice.h>
#include <cdevData.h>

void main()
   {
   cdevDevice *dir = cdevDevice::attachPtr("cdevDirectory");
   cdevData input, output;

   // *********************************************************
   // * Insert the name of the device and message to be resolved.
   // *********************************************************
   input.insert("device", "DEV1");
   input.insert("message", "get VAL");

   // *********************************************************
   // * Submit the request to the cdevDirectory object using the
   // * send command, and output the result (using asciiDump) if
   // * successful.
   // *********************************************************
   if(dir->send("serviceData", input, output)==CDEV_SUCCESS)
      {
      output.asciiDump(stdout);
      }
   }
```

"update"

The "update" message is submitted to the cdevDirectory object in order to add new or updated class definitions or device instances to the internal device definition data structures. The data that is submitted in the input cdevData object is in the same form as a corresponding entry in the CDEV DDL file.

The request is submitted to the cdevDirectory device through its *send*, *sendCallback*, or *sendNoBlock* interface. The following table shows the inputs that must be placed in the inbound cdevData object prior to submitting the command, and the outputs that can be retrieved when the command is completed.

Figure 50: cdevData input/output associated with a "update" message

	Tag Name	*Data Type*	*Content*
Message Input	***value*** or	character string	ASCII text definition of a DDL class or device instanciation.
	file	character string	Complete path to the file containing the updated DDL information.
Message Output	***value***	integer	Boolean completion status of the operation.

The following figure shows the source code necessary to insert the class definition for the "stdio" class into the device definition data structure using the cdevDirectory object.

Figure 51: Using the "update" message with a cdevDirectory device

```
#include <cdevSystem.h>
#include <cdevDevice.h>
#include <cdevData.h>

void main()
   {
   cdevDevice *dir = cdevDevice::attachPtr("cdevDirectory");
   cdevData input, output;

   // ***********************************************************
   // * Class definition to insert.
   // ***********************************************************
   char *def = "class stdio{verbs{get,set,monitorOn,monitorOff}}";

   // ***********************************************************
   // * Insert the class definition.
   // ***********************************************************
   input.insert("value", def);

   // ***********************************************************
   // * Submit the request to the cdevDirectory object.
   // ***********************************************************
   dir->send("update", input, output)
   }
```

"validate"

The "validate" message is submitted to the cdevDirectory object in order to verify that a combination of *device, DDL class, verb, attribute* or *message* represent a valid combination.

At least one of *device* or *DDL class* must be specified when using this message. If both are specified, the cdevDirectory object will confirm that the *device* is a member of the specified *DDL class*.

If *verb, attribute* or *message* are included, the cdevDirectory object will determine if each of them are members of the specified *device* and/or *DDL class.*

The request is submitted to the cdevDirectory device through its *send*, *sendCallback*, or *sendNoBlock* interface. The following table shows the inputs that must be placed in the inbound cdevData object prior to submitting the command, and the outputs that can be retrieved when the command is completed.

Figure 52: cdevData input/output associated with a "validate" message

	Tag Name	*Data Type*	*Content*
Message Input	***device*** or	character string	Name of the CDEV device.
	class	character string	Name of the DDL class.
	attribute	character string	Name of the attribute to validate (optional)

	message	character string	Name of the message to validate (optional)
	verb	character string	Name of the verb to validate (optional)
Message Output	***value***	integer	Boolean completion status of the operation.

The following figure shows the source code necessary to determine if the *verb* “monitorOn” is a member of the *class* “stdio”.

Figure 53: Using the “update” message with a cdevDirectory device

```
#include <cdevSystem.h>
#include <cdevDevice.h>
#include <cdevData.h>

void main()
   {
   cdevDevice *dir = cdevDevice::attachPtr(“cdevDirectory”);
   cdevData input, output;
   int result;

   input.insert(“class”, “stdio”);
   input.insert(“verb”, “monitorOn”);

   // ************************************************************
   // * Submit the request to the cdevDirectory object.
   // ************************************************************
   dir->send(“validate”, input, output)
   result = (int)output;
   fprintf(stdout, “monitorOn %s in stdio”, result?”IS”:”IS NOT”);
   }
```

8. Default Service Behavior for Standard Messages

Overview

The CDEV library is designed to provide a standard calling interface to dissimilar devices within a control system. This interface is accommodated through the use of the cdevDevice methods ***send, sendNoBlock*** and ***sendCallback***. However, because each service can define the names and behaviors of the messages that it supports, the user must be increasingly aware of which service may process its messages.

In order to reduce the required knowledge of the user, and to improve the consistency of all services, all CDEV services should provide well-defined support for a minimum list of *verbs*.

The following verbs should be implemented to provide a standard behavior in all CDEV services: ***get, set, monitorOn,*** and ***monitorOff.***

"get" Message

The *"get"* verb can be joined with any *attribute* of a device to form a "get" message. This message is then sent to the device in order to obtain the value of specified *properties* of the *attribute.* The following steps should be executed in order to utilize a *"get"* message.

1. Obtain a pointer to the *cdevDevice* object for the device that you wish to address.
2. Create a message string by concatenating the *attribute* that you wish to address to the *"get"* verb. For instance, to get the *VAL* attribute of a device, the message strings should be: ***"get VAL"***.
3. Optionally, use the cdevDevice object created in step 1 and the message string created in step 2 to obtain a pointer to a *cdevRequestObject* object.
4. Set the *context* of the *cdevDevice* or *cdevRequestObject* to indicate which properties you wish to obtain. A non-zero value in any property indicates that its value should be returned. If no context has been specified, the service should return the *value* property by default. A complete description of the context data object is provided in the cdevDevice documentation.
5. Use the ***send, sendNoBlock,*** or ***sendCallback*** message to submit the message to the device.
6. Evaluate the return value from the ***send*** command to determine if the operation completed successful. Any value other than ***CDEV_SUCCESS*** indicates that an error occurred in handling the message.
7. If the call was completed successfully, extract the desired properties from the resultant cdevData object.

"set" Message

The *"set"* verb can be joined with any *attribute* of a device to form a "set" message. This message is then sent to the device in order to set the value *property* of the *attribute.* The following steps should be executed in order to utilize a *"set"* message.

1. Obtain a pointer to the *cdevDevice* object for the device that you wish to address.
2. Create a message string by concatenating the *attribute* that you wish to address to the *"set"* verb. For instance, to set the *bdl* attribute of a device, the message string should be: ***"set bdl"***.
3. Optionally, use the cdevDevice object created in step 1 and the message string created in step 2 to obtain a pointer to a *cdevRequestObject* object.
4. Set the *value* property of the outbound cdevData object to the new value.
5. Use the ***send, sendNoBlock,*** or ***sendCallback*** message to submit the message to the device.
6. Evaluate the return value from the specific ***send*** command to determine if the message was transmitted successful. Any value other than ***CDEV_SUCCESS*** indicates that the message was not transmitted successfully.

In the following example, the *"get"* message will be used to obtain the properties *value, status and severity* from the *bdl* attribute of device *MQB1S01*, the *"set"* message will then be used to copy the *value* property to the *bdl* attribute of device *MQB1S02*.

Figure 54: Default behavior of the "get" and "set" messages

```
#include <cdevSystem.h>
#include <cdevDevice.h>
#include <cdevRequestObject.h>
#include <cdevData.h>

// ****************************************************************
// * The printError function will be used to output any error that
// * that occurs during the processing of the "get" or "set"
// * messages.
// ****************************************************************
int printError ( int errCode )
   {
   switch(errCode)
      {
      // ******* Unknown device or device/message mismatch ******
      case CDEV_INVALIDOBJ:
         printf("Unknown device or device/message mismatch\n");
         break;

      // * Communications error between application and service *
      case CDEV_NOTCONNECTED:
      case CDEV_IOFAILED:
      case CDEV_TIMEOUT:
         printf("Communications error while sending\n");
         break;
```

Figure 54: Default behavior of the “get” and “set” messages (continued)

```
      // ******* No data or bad data passed with message ********
      case CDEV_INVALIDARG:
      case CDEV_OUTOFRANGE:
      case CDEV_NOTFOUND:
      case CDEV_CONVERT:
         printf("Bad or missing value passed in message\n");
         break;

      // ******************** Generic Error ********************
      case CDEV_ERROR:
      case default:
         printf("Unable to send message\n");
         break;
      }
   }

void main()
   {
   cdevRequestObject * req1, *req2;
   cdevData            ctx;
   cdevData            output, input;
   double              val;
   int                 errorCode = CDEV_SUCCESS;

   // **********************************************************
   // * Obtain a pointer to the cdevRequestObject for the
   // * "get bdl" message on device "MQB1S01".
   // **********************************************************
   req1 = cdevRequestObject::attachPtr("MQB1S01", "get bdl");

   // **********************************************************
   // * Place a non-zero value in the properties value, status,
   // * severity.
   // **********************************************************
   ctx.set("value", 1);
   ctx.set("status", 1);
   ctx.set("severity", 1);

   // **********************************************************
   // * Set the context of the cdevRequestObject.
   // **********************************************************
   req1->setContext(ctx);

   // **********************************************************
   // * Submit the message to the device and test the return
   // * value to ensure that the message was processed correctly
   // **********************************************************
   if((errorCode=req1->send(NULL, &output))==CDEV_SUCCESS)
      {
      // **********************************************************
      // * Message was transmitted and processed successfuly.
      // **********************************************************
      char   stat[50], sev[255];
      output.get("value", &val);
```

Figure 54: Default behavior of the "get" and "set" messages

```
        output.get("status", stat, 50);
        output.get("severity", sev, 50);
        printf("Val:%f, status:%s, severity:%s", val, stat, sev);
        }
    else printError(errorCode);

    // ************************************************************
    // * Obtain a pointer to the cdevRequestObject for the
    // * "set bdl" message on device "MQB1S02".
    // ************************************************************
    req2 = cdevRequestObject::attachPtr("MQB1S02", "set bdl");

    // ************************************************************
    // * Insert the new value into the input cdevData object.
    // ************************************************************
    input.insert("value", value);

    // ************************************************************
    // * Submit the message "set bdl" to the device and test
    // * the return value to ensure that the message was
    // * processed successfully. Note that, by default, the set
    // * message does not generate any output.
    // ************************************************************
    if((errorCode==req2->send(&input, NULL))==CDEV_SUCCESS)
        {
        // *******************************************************
        // * Message was transmitted and processed successfuly.
        // *******************************************************
        printf("Message was transmitted successfully\n");
        }
    else printError(errorCode);
    }
```

"monitorOn" Message

The *"monitorOn"* verb can be joined with any *attribute* of a device to form a "*monitorOn*" message. This message is tells the cdevDevice that each time one of the monitored *properties* changes, it should call the user specified callback function with the updated value. This message should always be submitted using the **sendCallback** method of the cdevDevice or cdevRequestObject in order to provide the address of the callback function.

The following steps should be executed in order to submit a *"monitorOn"* message.

1. Obtain a pointer to the *cdevDevice* object for the device that you wish to address.

2. Create a message string by concatenating the *attribute* that you wish to address to the *"monitorOn"* verb. For instance, to monitor a property of the *bdl* attribute of a device, the message string should be: ***"monitorOn bdl"***.

3. Optionally, use the cdevDevice object created in step 1 and the message string created in step 2 to obtain a pointer to a *cdevRequestObject* object.

4. Set the *context* of the cdevDevice or cdevRequestObject to indicate the *properties* that you wish to monitor, and the *properties* you wish to receive when a change

occurs. The context used by the "*monitorOn*" message differs from that of the *"get"* message in that different non-zero values have special meanings. The following is a list of integer values that should be supported by the context of a "*monitorOn*" message.

0	The value of this property should never be returned.
1	This property should not be monitored, but it should be returned as context when a monitored value changes; e.g. a time stamp.
2	Monitor this property and call the callback function when it changes, however, return only this property in the resultant cdevData object.
3	Monitor this property and call the callback function when it changes and include the value of all properties whose context is 1 in the resultant cdevData object.

5. Create a *cdevCallback* object that contains the address of the cdevCallback function and a void pointer to any user argument that should be provided to the callback function.

6. Use the ***sendCallback*** message to submit the message to the device.

7. Evaluate the return value from the specific ***sendCallback*** command to determine if the message was transmitted successful. Any value other than ***CDEV_SUCCESS*** indicates that the message was not transmitted successfully.

8. Monitor the status returned to the *cdevCallbackFunction* to determine if the callback is operating correctly. Any of the following values may be returned in the *status* parameter.

CDEV_SUCCESS:	The value has changed and the new value is stored in the cdevData parameter. This status is also generated when the monitorOn command is initially issued.
CDEV_DISCONNECTED:	The connection has been lost between the application and the server.
CDEV_RECONNECTED:	The connection between the application and the server has been re-established and the updated value is available in the cdevData parameter.
CDEV_ERROR:	The message was not processed successfully.
CDEV_INVALIDOBJ:	Invalid cdev request object used.
CDEV_INVALIDARG:	Invalid argument passed to cdev call.
CDEV_INVALIDSVC:	Wrong service during dynamic loading.
CDEV_NOTCONNECTED:	Not connected to low level network service.
CDEV_IOFAILED:	Low level network service IO failed.

CDEV_CONFLICT:	Conflicts of data types or tags.
CDEV_NOTFOUND:	Cannot find specified data in cdevData.
CDEV_TIMEOUT:	Time out.
CDEV_CONVERT:	cdevData conversion error.
CDEV_OUTOFRANGE:	Value out of range for device attribute.
CDEV_NOACCESS:	Insufficient access to perform request.
CDEV_ACCESSCHANGED:	Change in access permission of device.

9. Because the monitorOn command is a single message that may generate many responses, the developer may have difficulty in determining when the last reply has been received. In order to accomodate this, the developer may call the cdevCallback::isTransactionDone() method to determine if the callback that is being processed is the last one that is associated with that request. The service developer is expected to use the cdevCallback::fireCallback method to ensure that this parameter is set to the proper value.

"monitorOff" Message

The *"monitorOff"* verb can be joined with any *attribute* of a device to form a "*monitorOff*" message. This message is tells the cdevDevice to deactivate a monitorOn command that was previously set on one or more of its attributes.

This message should always be called using the *sendCallback* method in order to specify the address of the callback function that was used in creating the monitor.

The service checks the following things to determine which monitor is to be removed.

device	The service will locate all active monitors that have been placed on the specified device.
attribute	From the list of obtained above, the service will locate all active monitors on the specified attribute.
function	From the list obtained above, the service will locate all active monitors that have the specified callback function. If this value is NULL, then all elements from the previous list will be removed.
userarg	From the list obtained above, the service will locate and remove all monitors that have the same user argument. If this value is NULL, then all elements from the previous list will be removed.

The following steps should be executed in order to submit a *"monitorOff"* message.

1. Obtain a pointer to the *cdevDevice* object for the device that you wish to address.
2. Create a message string by concatenating the *attribute* that you wish to address to the *"monitorOff"* verb. For instance, to monitor a property of the *bdl* attribute of a device, the message string should be: ***"monitorOn bdl"***.
3. Optionally, use the cdevDevice object created in step 1 and the message string created in step 2 to obtain a pointer to a *cdevRequestObject* object.
4. Create a *cdevCallback* object that contains the address of the cdevCallback

function and a void pointer to the user argument that was originally provided to the monitorOn message.

5. Use the ***sendCallback*** message to submit the message to the device.

The following example illustrates how to use the *"monitorOn"* and *"monitorOff"* to install and remove monitors. This example also demonstrates the to correct way to utilize the cdevCallback function.

Figure 55: Default behavior of the "monitorOn" and "monitorOff" messages

```
#include <cdevSystem.h>
#include <cdevDevice.h>
#include <cdevRequestObject.h>
#include <cdevData.h>
#include <cdevCallback.h>

// *****************************************************************
// * This is the callback function that will be executed each time
// * one of my monitored values changes.
// *****************************************************************
void callback (int status, void * userarg,
              cdevRequestObject & req, cdevData & data)
   {
   switch(status)
      {
      // *********************************************************
      // * If I receive one of these message then I know that
      // * updated data is available in the cdevData object.
      // *********************************************************
      case CDEV_SUCCESS:
      case CDEV_RECONNECTED:
         // *****************************************************
         // * I will call the getType method to determine if the
         // * value attribuite is present in the cdevData object.
         // * If it is, then I will known that it is responsible
         // * for triggering this callback.
         // *****************************************************
         if(data.getType("value")!=CDEV_INVALID)
            {
            // *************************************************
            // * I have specified that I wanted status and
            // * severity to be returned as context whenever the
            // * value property has changed.  Therefore, I will
            // * output all of these properties.
            // *************************************************
            double * valPtr;
            char   * statPtr;
            char   * sevPtr;
            data.find("value",    (void *&)valPtr);
            data.find("status",   (void *&)statPtr);
            data.find("severity", (void *&)sevPtr);
            printf("Val:%f, status:%s, severity:%s",
                   *valPtr, statPtr, sevPtr);
            }
         // *****************************************************
         // * I will call the getType method of the cdevData
         // * object to determine if the controlHigh property is
         // * present. If it is, I will know that it triggered
         // * this callback.
         // *****************************************************
```

Figure 55: Default behavior of the “monitorOn” and “monitorOff” messages (continued)

```
        else if(getType("controlHigh")!=CDEV_INVALID)
            {
            double * controlPtr;
            data.find("controlHigh", (void *&) controlPtr);
            printf("Control High is now %f\n", *controlPtr);
            }
    break;

    // ************************************************************
    // * This status indicates that the connection has been lost
    // * between the application and the server.
    // ************************************************************
    case CDEV_DISCONNECTED:
        fprintf(stderr, "Connection lost to server\n");
        break;

    // ************************************************************
    // * This status indicates that an error occurred while
    // * installing or operating the monitor.
    // ************************************************************
    case CDEV_ERROR:
        fprintf(stderr, "Error while monitoring");
        break;
    }
}

void main()
    {
    cdevSystem        * system;
    cdevDevice        * dev;
    cdevRequestObject * req;
    cdevData            ctx;

    // ************************************************************
    // * Construct a cdevCallback object that has a pointer to the
    // * cdevCallback function and a NULL user argument.
    // ************************************************************
    cdevCallback cb(callback, NULL);

    // ************************************************************
    // * Obtain a pointer to the default cdevSystem in order to
    // * accomodate polling.
    // ************************************************************
    system = &cdevSystem::defaultSystem();

    // ************************************************************
    // * Obtain a pointer to the cdevDevice MQB1S01
    // ************************************************************
    dev = cdevDevice::attachPtr("MQB1S01");

    // ************************************************************
    // * Obtain a pointer to the cdevRequestObject for the
    // * "monitorOff bdl" message on device "MQB1S01".
    // ************************************************************
    req = dev->getRequestObject("MQB1S01", "monitorOn bdl");
```

Figure 55: Default behavior of the “monitorOn” and “monitorOff” messages (continued)

```
// ############################################################
// # Setup the context to indicate the data that should be
// # returned whenever one of the values is changed.
// ############################################################

// ************************************************************
// * Any time the “value” property changes, I want to receive
// * the “status” and “severity” properties.
// ************************************************************
ctx.insert(“value”, 3);
ctx.insert(“status”, 1);
ctx.insert(“severity”, 1);

// ************************************************************
// * Any time the “controlhigh” property changes, I only want
// * to receive its new value.
// ************************************************************
ctx.insert(“controlHigh”, 2);

// ************************************************************
// * Set the context of the cdevRequestObject to the new
// * context.
// ************************************************************
req->setContext(ctx);

// *************************************************************
// * Submit the “monitorOn” message using the sendCallback
// * method of the cdevRequestObject.
// *************************************************************
if(req->sendCallback(NULL, cb)==CDEV_SUCCESS)
    {
    // *********************************************************
    // * Pend for 60 second.
    // *********************************************************
    system->pend(60.0);

    // *********************************************************
    // * Send a “monitorOff bdl” message to disable the monitor
    // * that was just installed.
    // *********************************************************
    dev->sendCallback(“monitorOff bdl”, NULL, cb);
    }
else fprintf(stderr, “Failed to install monitor\n”);
}
```