Service Developer's Guide

Techniques for Developing Services 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

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

VxWorks is a register trademark of Wind River Systems.

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

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

Ultrix and DEC are registered trademarks of Digital Equipment Corporation.

HPUX is a registered trademark of Hewlett Packard.

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

Table of Contents

List of Figures

1. Overview of cdev Services

What is a cdev Service

The cdev library defines a command set that can be used to provide a homogenous interface to differing control systems. In order to incorporate a control system into the cdev environment, the developer must create *linking code* that allows the cdev system to communicate with the system. This *linking code* is called a *cdev service.* At a minimum, the cdev service developer must develop two interface classes and a simple constructor function in order to create a new service. These classes and functions will be described at length in the sections that follow.

Service Classes

Each service must define a service class that provides the mechanisms for communicating with the underlying control system. This class is inherited from the *cdevService* class, from which it gains most of its functionality. The main effort for the developer in creating this class is in the *flush, poll and pend* methods. These methods are functionally identical to those defined in the *cdevSystem* object, however, the developer must provide concrete mechanisms for communicating with the intended control system, as well as managing and reporting errors that might occur during normal operation.

Request Object Classes

The service developer must also create a request object class that provides the mechanisms for a *cdevDevice* object to communicate with the developer's service class. This class is inherited from the *cdevRequestObject* class, from which it gains most of its functionality. The primary effort for the developer in creating this class is in the *send, sendNoBlock* and *sendCallback* functions. The service creator must also develop his strategy for efficiently communicating with the underlying system. The source code in the following chapters will illustrate a queueing scheme that provides a very simple and efficient linkage between the service and the request object. The request object is also responsible for detecting and reporting errors that occur within its domain.

Loader Functions

The loader function is an extern "C" function that is called by the cdevSystem object to dynamically construct a copy of the service class.

What the Developer Must Understand

In order to create cdev services, the developer must have an extensive understanding of the complete cdev system. A potential service developer should first develop applications using existing cdev services in order to understand the expected behavior of a service. The developer should then familiarize himself completely with all of the classes within the cdev library, paying special attention to the *cdevService* class and the *cdevRequestObject* class.

The following chapters describe the structure and syntax of the most important of these classes and an overview of how certain messages are expected to behave. This manual finishes with the complete annotated source code for a demonstration service that operates on a virtual control system.

2. Developing cdev Services

Steps in Developing a cdev Service

1. **Install and build the cdev distribution.** Obtain a copy of the most recent cdev distribution and install it on your system.

2. **Define the devices in your control system.** Define the names, attributes and messages associated with each device in your control system. This information will be critical in the construction of the *Device Definition Language* file that cdev will use to determine which service will process messages for a device.

3. **Design and document specific device/message behavior.** Determine the specific inputs and outputs required for each device to process a message. Design and document how the service will route messages to the underlying control system. Ensure that the disposition of standard messages within your service is consistent with the behavior of standard cdev services.

4. **Develop your cdevService object.** This object is a sub-class of the *cdevService* class and will be responsible for responding to *flush, poll,* and *pend* requests from the cdevSystem object. The service object should be able to perform all interface tasks necessary to link the service specific *cdevRequestObjects* to the underlying control system.

5. **Develop your cdevRequestObject object.** This object is a sub-class of the *cdevRequestObject* class and will be responsible for responding to *send, sendNoBlock* and *sendCallback* requests from the cdev application.

6. **Develop your cdevCollectionRequest object (optional).** If your service will provide specialized support for collections of devices, it wil be necessary to develop a cdevCollectionRequest object that processes these requests.

7. **Create a service loader function.** This function is described earlier in this document and is used by the cdevSystem object to an instantiate a new service object for this service.

8. **Compile and link the shared object.** Compile a position-independent shared object file that cdev can load on request. The service's shared object file should contain all of the object code necessary to directly load and utilize the service. This file should be copied to the directory where the other cdev services are stored.

3. The cdevService Class

Overview of the cdevService Class

The cdevService C++ class is an abstract base class for all cdev services. This class defines the mechanisms that cdev will utilize to communicate with your underlying control system. It is the responsibility of the service developer to '*flesh out*' the virtual functions that are defined within this class and to develop the code necessary to communicate with the underlying control system.

The primary methods that developers must concern themselves with are *flush, poll pend,* and *getRequestObject.* These methods represent the majority of the work that must be performed in developing a cdev service.

The *flush* Method

The *flush* method is responsible for submitting any unsent messages to the device. This may entail submission of a message using a network protocol or simply calling a statically linked C function. The *cdevSystem* object will call this method prior to each *pend* or *poll* operation, or whenever the *flush* method of the system object is called directly by the user.

The *poll* Method

The *poll* method is responsible for directly polling each of the physical devices that are managed by the service to detect if they require attention (typically by checking a single socket). This method is typically utilized to allow the physical device an opportunity to return a response to a previously sent message. The *cdevSystem* object will call this method whenever the application calls the *poll* method of the system object. Most services can route this call directly to their associated *pend* method.

The *pend* Method

The *pend* method allows the service to wait for a period of time for one of its underlying devices to require attention. After waiting for a specified period of time, this method will return *CDEV_SUCCESS* if it successfully serviced any of its underlying devices, or *CDEV_TIMEOUT* if no device became active during that period. This method is typically called by the *cdevSystem* object in response to a change in state of one or more of the service's file descriptors.

The *getRequestObject* Method

The *getRequestObject* method is used by the cdevSystem object to obtain a new instance of one of the service's request objects in response to a request made by the application. The service may internally define many request object types for use by applications, therefore, it is the responsibility of this function to return the correct request object for the specified *device* / *message* combination. Typically a service will only define one type of request object and will use it for all requests.

Public Member Functions of the cdevService Class

Public Functions of the cdevService Class

flush

int flush (void);

Flushes any pending outbound requests to the appropriate servers. This is a pure virtual function that must be provided by the service developer.

poll *int poll (void);*

Directly polls each of the cdevService's underlying file descriptors for activity, and delivers any asynchronous callbacks that are ready. This is a pure virtual function that must be provided by the service developer.

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 cdevService 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 cdevService object will call the appropriate function to process the event and dispatch any asynchronous callbacks that are ready. This is a pure virtual function that must be provided by the service developer.

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 cdevService 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 cdevService object will call the appropriate functions to process the event and dispatch any asynchronous callbacks that are ready. This is a pure virtual function that must be provided by the service developer.

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

Obtains a pointer to a *cdevRequestObject* that is specific to this service and the specified device/message combination. This method should only be called by the *cdevSystem* object. This is a pure virtual function that must be provided by the service developer.

getNameServer *int getNameServer (cdevDevice* &server);*

Obtains a cdevDevice object identifying the name server for this service. A service is not required to provide its own name server and may simply set the *server* parameter to *NULL*. This is a pure virtual function an must be provided by the service developer.

getCollectionRequest *int getCollectionRequest (char ** devices, int nDevices, char * msg, cdevCollectionRequest * &req);*

This method allows the caller to obtain a cdevCollectionRequest object that will contain only devices that are associated with the service. A default mechanism is provided to support this functionality, however, the developer may create a special request object to optimize these operations.

getFd

int getFd (int &fd, int &numFd);*

Retrieves a list of file descriptors that are contained within the cdevService object. The *fd* pointer will be given the pointer to the internal array of file descriptors, and the *numFD* parameter will be set to the number of file descriptors in the list. A service that does not use file descriptors should set the *fd* and *numFD* parameters to *NULL* and *0*, respectively. This is a pure virtual function that must be provided by the service developer.

registerFd

int registerFd (int fd, int opened);

The service developer may implement this method to allow external file descriptors to be added to the list of file descriptors in the service. The *fd* parameter should contain the file descriptor, and the *opened* parameter should contain 1 to add it to the list or 0 to remove it from the list.

autoErrorOn

int autoErrorOn (void);

Informs the cdevService 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 cdevService object.

autoErrorOff

int autoErrorOff (void);

Informs the cdevService 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 cdevService object.

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 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.

name

*char *name (void) const;*

Retrieves the name of the service. This method is used extensively to determine the service that underlies a specific *cdevRequestObject* object. If not over-ridden by the service developer, this method will return the string “cdevService”.

4. The *cdevRequestObject* Class

Overview of the cdevRequest Object Class

The *cdevRequestObject* C++ class is the application's interface to the underlying service. Each service must provide a service specific request object that applications may use to send messages to the service.

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. The majority of work in the development of a service specific request object is in the development of these three methods.

Public Member Functions of the cdevRequest Object 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 accommodate 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 accommodate 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 1: Return codes generated by the send method.

```
CDEV_WARNING:         The failure was non-critical.
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_INVALIDOP:       The operation is not supported.
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 must 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 specified 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.

5. The cdevCollectionRequest Class

Overview of the cdevCollection Request Class

The cdevCollectionRequest class is an abstract base class from which other cdevCollectionRequest objects are derived. It provides a protected constructor and destructor that are used to initialize its internals, however, the primary mechanism that is used to obtain a cdevCollectionRequest object is the *attachPtr* method.

Because the cdevCollectionRequest is inherited from the cdevRequestObject, all of the methods of that class must be fulfilled in addition to those specific to the cdevCollectionRequest class.

Public Member Functions of the cdevCollection Request Class

constructor

*cdevCollectionRequest(char **devices, int nDevices,*
*char * msg, cdevSystem & system);*

This is the constructor for the cdevCollectionRequest class. It has the following properties.

- This method is ***protected*** to prevent the direct instantiation of new *cdevCollectionRequests*. New instances of the *cdevCollectionRequest objects are* created by using the *attachPtr* or *attachRef* method of the *cdevRequestObject* class which will call the local *attachPtr* method to create a new object if necessary.

- The constructor is called by the cdevCollection object and is provided with a list and count of devices that will be included in the collection and the message that will be sent to them.

- The *cdevSystem* reference that is provided is the *cdevSystem* instance that will be used to *poll*, *pend* and *flush* the *cdevCollectionRequest* object.

destructor

virtual ~cdevCollectionRequest (void);

This is the destructor for a cdevCollectionRequest object. It has the following properties.

- This method is protected to prevent the *cdevCollectionRequest* object from being destroyed by the application. This method should only be called by the *cdevSystem* object when the application is terminating.

- Because the *cdevCollectionRequest* object will normally be referred to as a *cdevRequestObject* object, this destructor is virtual to ensure that the ***'most senior'*** destructor is called first.

attachPtr

*cdevCollectionRequest * attachPtr*
*(cdevCollection &col, char *msg, cdevSystem &sys);*

This method is used by the cdevCollection object to obtain a new cdevCollectionRequest object.

- This method will obtain a copy of the device names from the cdevCollection object and will poll the cdevDirectory object to determine which service each of them is associated with.
- If the devices are all from a single service, this method will return a service specific collection request object.
- If the devices are from a variety of services, this method will return a cdevGrpCollectionRequest that contains the service specific collection request objects.
- Device/message combinations that are not associated with a service will be ignored.
- If none of the device/message combinations can be associated with a service, then an error message will be generated and NULL will be returned.

className

*char * className (void);*

This method returns the name of the class; "*cdevCollectionRequest*". If the developer inherits a service specific cdevCollectionRequest, then this method should not be altered or overridden.

resultCodeTag

int resultCodetag (void);

This method returns the integer tag that should be used to insert the result code that was geneterated when the message was sent to the actual device.

6. The Service Loader Function

Overview

The service loader function is used to create an initial instance of the service after the its associated shared object file has been dynamically loaded.

Naming Convention

The naming convention for this class is very specific. The following syntax is required:

Figure 2: Naming convention and syntax for the cdevService loader function

```
cdevService *newXxxxxService (char * name, cdevSystem * system);
```

In the function name above the *Xxxxx* should be replaced by the name of the service with the first character capitalized. A service named *demo* would have a loader function named *newDemoService.* The loader function is only required to allocate a new service class for the service and return a pointer to it. The following example illustrates how the loader function for the demo service would be written.

Figure 3: Example cdevService loader function

```
#include <cdevService.h>

// *****************************************************************
// * Include file with definition of the service class
// *****************************************************************
#include “demoService.h”

// *****************************************************************
// * Loader function for the demoService class
// *****************************************************************
cdevService *newDemoService (char * name, cdevSystem * system)
    {
    return new demoService(name, *system);
    }
```

7. The cdevTranObj Class

Overview of the cdevTranObj Class

The cdevTranObj (*cdev transaction object*) C++ class is a container class that is used to maintain information that is required for individual operations within a service. This class stores a copy of the *cdevRequestObject* that was used to submit a request, the *cdevSystem* object in which the request object was created, the *cdevCallback* object provided by the user, and the *cdevData* object that the returned data should be placed in. Because this object is intended for usage only by cdev internals, all of its data elements are *public*.

The *cdevTranObj* is used to submit a request from the cdevRequestObject to the cdevService. It is also used to place a transaction into a group of requests using the *cdevGroup* object. The service notifies the *cdevSystem* object and the *cdevGroup* objects that this transaction has been processed by calling its *removeFromGrps* method. This will effectively remove the *cdevTranObj* from all groups that it is associated with.

Public Data Properties of the cdevTranObj Class

system_

*cdevSystem *system_;*

This is a pointer to the cdevSystem that contains this transaction object. The transaction will be processed whenever the *poll* or *pend* methods of this *cdevSystem* object are executed.

reqObj_

*cdevRequestObject *reqObj_;*

This is a pointer to the cdevRequestObject that was called to submit this transaction. This *cdevRequestObject* will be submitted to the user specified callback function when the transaction has been completed.

resultData_

*cdevData *resultData_;*

This is the cdevData object that will be populated with the results of the transaction. This object will also be submitted to the user specified callback function when the transaction has been completed.

userCallback_

*cdevCallback *userCallback_;*

This class contains a pointer to the caller specified callback function as well as any user argument. The callback function will be executed when the transaction has been completed. Note that it is the responsibility of the service to execute this callback function when it has finished processing the transaction.

Public Member Functions of the cdevTranObj Class

status

int status (void);

Returns the status of the *cdevTranObj* object. The value returned will be **1** if this object is a member of any *cdevGroup* object, or **-1** if this object is not a member of any *cdevGroup* object.

removeFromGrps *int removeFromGrps (void);*

Removes this transaction object from any *cdevGroup* object that it may be in.

enableDeleteCbk *void enableDeleteCbk (void);*

Sets the internal flag telling the transaction object to delete its internal callback object while executing its destructor. This is the default behavior for the cdevTranObj.

disableDeleteCbk *void disableDeleteCbk (void);*

Sets the internal flag telling the transaction object not to delete its internal callback object while executing its destructor. This mode should be used when the user specified callback object is shared by numerous transaction objects.

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 4: 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 4: 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);
        output.get("status", stat, 50);
```

Figure 4: Default behavior of the "get" and "set" messages

```
        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 5: 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 5: 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 5: 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");
}
```

9. *demoService*: A Sample cdev Service

Overview of the demoService

The *demoService* service is a sample cdev service that operates on a virtual control system. For the purposes of this illustration, this control system is represented by a group of demoDevice classes.

The demoDevice Object

Each demoDevice object represents a single device that has two value attributes: ***BDL*** and ***VAL.*** Each of these two attributes has the following properties:

1. ***value:*** This is the double precision value of the attribute.
2. ***status:*** This is the current status of the attribute, it may have one of the following values.

   0 No error.
   1 The *value* property is below the range specified by the *controlLow* property.
   2 The *value* property is below the range specified by the *alarmLow* property.
   3 The *value* property is above the range specified by the *alarmHigh* property.
   4 The *value* property is above the range specified by the *controlHigh* property.

3. **severity:** This is a string that describes the current *status* of the attribute.
4. **units:** These are the units that are used to express the value properties of the attribute.
5. **controlLow:** This is a double precision number reflecting the minimum physical value for the attribute.
6. **controlHigh:** This is a double precision number reflecting the maximum physical value for the attribute.
7. **alarmLow:** This is a double precision number reflecting the lower alarm threshold for this attribute.
8. **alarmHigh:** This is a double precision number reflecting the upper alarm threshold for this attribute.

The complete source code for the demoDevice class and its instances are provided in the following source code examples.

Figure 6: demoDevice.h: Header file for devices used by the demoService

```
#ifndef _DEMO_DEVICE_H_
#define _DEMO_DEVICE_H_

#include <stdlib.h>

enum demoStatus {
   NORMAL=0, ERROR_LOW, ALARM_LOW, ALARM_HIGH, ERROR_HIGH };

// ****************************************************************
// * struct demoAttrib:
// *The demoAttrib structure contains all of the properties
// *that are supported by a single attribute of a demoDevice
// *structure.
// ****************************************************************
typedef struct
   {
   charname     [32];// Name of this attribute
   charunits    [32];// Units that value is measured in
   doublevalue;// Value of the attribute
   doublecontrolLow;// Minimum value of the attribute
   doublecontrolHigh;// Maximum value of the attribute
   doublealarmLow;// Minimum value resulting in alarm
   doublealarmHigh;// Maximum value resulting in alarm
   } demoAttrib;


// ****************************************************************
// * class demoDevice:
// *The demoDevice structure is a crude representation of a
// *control system device.  Although these devices are not
// *attached to physical hardware, they support the concept of
// *value, units, control limits, display limits, alarm limits,
// *alarm status and alarm severity.
// *
// *The demoService is designed to provide a cdev interface to
// *these devices.  Note that this device knows nothing about
// *cdev or any component of cdev.  It is entirely the
// *responsibility of cdev to accomodate the device with little
// *or no alterations.
// ****************************************************************
class demoDevice
{
private:
   char       *name;// Name of this device
   demoAttrib *attr;// List of embedded attributes
   size_tcount;// Number of embedded attributes

public:
   demoDevice ( char * Name, demoAttrib * Attr, size_t Count );
   ~demoDevice( void );

   char * getName( void );// Get name of the device
   size_t getCount( void );// Get number of attributes
   int    getIndex( char *Attr );// Get index of Attr
   char * getUnits( int idx );// Get units of measure
```

Figure 6: demoDevice.h: (continued)

```
    double getValue( int idx );// Get attribute value
    double getControlLow( int idx );// Get min control value
    double getControlHigh( int idx );// Get max control value
    double getAlarmLow( int idx );// Get min alarm value
    double getAlarmHigh( int idx );// Get max alarm value
    demoStatus getStatus( int idx );// Get alarm status
    char *     getSeverity( int idx );// Get alarm severity

    int setValue ( int idx, double val );// Set the attribute value
};

// ****************************************************************
// * These are the names of the demoDevices that will exists in
// * this demonstration control system.
// ****************************************************************
extern demoDevice DEVICE0;
extern demoDevice DEVICE1;
extern demoDevice DEVICE2;
extern demoDevice DEVICE3;
extern demoDevice DEVICE4;

#endif /* _DEMO_DEVICE_H_ */
```

Figure 7: demoDevice.cc: C++ source for the demoDevice class

```
#include <string.h>
#include "demoDevice.h"

// ****************************************************************
// * demoDevice::demoDevice :
// * This is the constructor for the demoDevice.  It assigns the
// * value passed in the Name parameter as the name of the device,
// * and it uses the array provided in the Attr parameter as its
// * list of attributes.  The number of attributes in the array
// * should be provided in the Count parameter.
// ****************************************************************
demoDevice::demoDevice (char *Name, demoAttrib *Attr, size_t Count)
    : attr(Attr), count(Count)
    {
    name = Name?strdup(Name):NULL;
    }

// ****************************************************************
// * demoDevice::~demoDevice :
// * This is the destructor for the demoDevice, it deletes the name
// * string that was allocated in the constructor.
// ****************************************************************
```

Figure 7: demoDevice.cc: (continued)

```
demoDevice::~demoDevice ( void )
    {
    if(name!=NULL) delete name;
    }

// ****************************************************************
// * demoDevice::getName :
// * This function allows the caller to obtain the name of the
// * demoDevice
// ****************************************************************
char * demoDevice::getName ( void )
    {
    return name;
    }

// ****************************************************************
// * demoDevice::getCount :
// * This function returns the number of attributes in this
// * demoDevice.
// ****************************************************************
size_t demoDevice::getCount ( void )
    {
    return count;
    }

// ****************************************************************
// * demoDevice::getIndex :
// * This method allows the caller to obtain the index of a specific
// * attribute by name.  If the value returned is less than 0, then
// * the attribute does not exist on this device.
// ****************************************************************
int demoDevice::getIndex ( char * Attr )
    {
    for(int i=count-1; i>=0 && strcmp(Attr, attr[i].name)!=0; i--);
    return i;
    }

// ****************************************************************
// * demoDevice::getUnits :
// * This method returns the units of measure used by the attribute
// * at the specified index.  Returns NULL if the index is out of
// * range.
// ****************************************************************
char * demoDevice::getUnits ( int idx )
    {
    return (idx>=0 && idx<count)?attr[idx].units:NULL;
    }

// ****************************************************************
// * demoDevice::getValue :
// * Allows the user to obtain the value of the specified attribute.
// * Returns 0 if the attribute is invalid.
// ****************************************************************
double demoDevice::getValue ( int idx )
```

Figure 7: demoDevice.cc: (continued)

```
    {
    return (idx>=0 && idx<count)?attr[idx].value:0.0;
    }

// ****************************************************************
// * demoDevice::getControlLow :
// * Returns the minimum legal value for the device.  Returns -1E300
// * if the attribute is invalid.
// ****************************************************************
double demoDevice::getControlLow ( int idx )
    {
    return (idx>=0 && idx<count)?attr[idx].controlLow:-1E300;
    }

// ****************************************************************
// * demoDevice::getControlHigh :
// * Returns the maximum legal value for the device.  Returns 1E300
// * if the attribute is invalid.
// ****************************************************************
double demoDevice::getControlHigh ( int idx )
    {
    return (idx>=0 && idx<count)?attr[idx].controlHigh:1E300;
    }

// ****************************************************************
// * demoDevice::getAlarmLow :
// * Returns the minimum legal value for the device.  Returns -1E300
// * if the attribute is invalid.
// ****************************************************************
double demoDevice::getAlarmLow ( int idx )
    {
    return (idx>=0 && idx<count)?attr[idx].alarmLow:-1E300;
    }


// ****************************************************************
// * demoDevice::getAlarmHigh :
// * Returns the maximum legal value for the device.  Returns 1E300
// * if the attribute is invalid.
// ****************************************************************
double demoDevice::getAlarmHigh ( int idx )
    {
    return (idx>=0 && idx<count)?attr[idx].alarmHigh:1E300;
    }

// ****************************************************************
// * demoDevice::getStatus :
// * Calculates and returns the alarm status of the attribute at the
// * specified index.  Returns NULL if the index is out of range.
// ****************************************************************
demoStatus demoDevice::getStatus ( int idx )
    {
    demoStatus status = NORMAL;
    if(idx>=0 && idx<count)
        {
```

Figure 7: demoDevice.cc: (continued)

```
        if(attr[idx].value<=attr[idx].controlLow)
            status = ERROR_LOW;
        else if(attr[idx].value<=attr[idx].alarmLow)
            status = ALARM_LOW;
        else if(attr[idx].value<attr[idx].alarmHigh)
            status = NORMAL;
        else if(attr[idx].value<attr[idx].controlHigh)
            status = ALARM_HIGH;
        else status = ERROR_HIGH;
        }
    return status;
    }

// *****************************************************************
// * demoDevice::getSeverity :
// * Calculates and returns the alarm severity of the attribute at
// * the specified index.  Returns NULL if the index is out of
// * range.
// *****************************************************************
char * demoDevice::getSeverity ( int idx )
    {
    static char * status[] =
        {
        "No alarm",
        "Attribute value is below its minimum legal boundary",
        "Attribute value is at or below its minimum alarm boundary",
        "Attribute value is at or above its maximum alarm boundary",
        "Attribute value is above its maximum legal boundary"
        };
    return status[getStatus(idx)];
    }

// *****************************************************************
// * demoDevice::setValue :
// * Allows the user to set the value of the specified attribute.
// * Returns 0 if the attribute was successfully set, otherwise -1.
// *****************************************************************
int demoDevice::setValue ( int idx, double val )
    {
    int result = -1;

    if(idx>=0 && idx<count)
        {
        if(val > attr[idx].controlLow &&
           val < attr[idx].controlHigh )
            {
            attr[idx].value = val;
            result = 0;
            }
        }
    return result;
    }
```

Figure 7: demoDevice.cc: (continued)

```
// ***************************************************************
// * These are the attribute definitions that will be used to
// * construct the fictitious devices in the demo control system.
// * The values and names used here are inconsequential.
// ***************************************************************
demoAttrib DEVICE0Attrib [2] =
    {
    { "VAL", "amps",         0.0, -30.0, 30.0, -20.0, 20.0 },
    { "BDL", "gauss-meters", 0.0, -14.0,  7.0, -10.0,  5.0 }
    };

demoAttrib DEVICE1Attrib [2] =
    {
    { "VAL", "amps",         0.0, -20.0, 20.0, -15.0, 15.0 },
    { "BDL", "gauss-meters", 0.0, -10.0, 10.0,  -7.5,  7.5 }
    };

demoAttrib DEVICE2Attrib [2] =
    {
    { "VAL", "amps",         0.25, -12.0, 10.0, -9.0,  7.0 },
    { "BDL", "gauss-meters", 1.0,  -10.0, 10.0, -7.5,  7.5 }
    };

demoAttrib DEVICE3Attrib [2] =
    {
    { "VAL", "amps",         1.25, -12.0, 10.0, -9.0,  7.0 },
    { "BDL", "gauss-meters", 8.0,  -10.0, 10.0, -7.5,  7.5 }
    };

demoAttrib DEVICE4Attrib [2] =
    {
    { "VAL", "amps",         11.25, -12.0, 13.0, -9.0, 10.0 },
    { "BDL", "gauss-meters",  8.0,  -10.0, 10.0, -7.5,  8.25 }
    };

// ***************************************************************
// * Instances of the ficticious devices in out control system.
//****************************************************************
demoDevice DEVICE0("DEVICE0", DEVICE0Attrib, 2);
demoDevice DEVICE1("DEVICE1", DEVICE1Attrib, 2);
demoDevice DEVICE2("DEVICE2", DEVICE2Attrib, 2);
demoDevice DEVICE3("DEVICE3", DEVICE3Attrib, 2);
demoDevice DEVICE4("DEVICE4", DEVICE4Attrib, 2);
```

The demoService Object

The *demoService* C++ class is derived from the *cdevService* class. It maintains a list of references to the devices that are defined in the *demoDevice* source code. This class also provides the mechanisms that its request objects will use to communicate with these devices.

The enqueueTransaction Method

The *demoService* class uses a queueing scheme to manage outbound transaction objects. Whenever a request object wishes to submit a message to the service, it will call the service's *enqueueTransaction* method. This method receives a pointer to the *cdevTranObj* object that contains information about the transaction. It also receives a pointer to a cdevData object that contains any information that the service will need to complete the request.

Messages passed to the *enqueueTransaction* method will be stored in an internal linked-list until the *pend* or *poll* methods are called. The *pend* and *poll* methods are responsible for transferring the messages to the *processTransaction* method.

The processTransaction Method

The *processTransaction* method is where all messages sent to the service are dispatched to the underlying *demoDevice* control system. Unlike the *pend* and *poll* methods which restrict their activity to a caller specified period of time, the *processTransaction* method operates without regard to time. It will process one complete message before returning. Because this is a private member function, it may only be called by methods that exist within the class.

The newDemoService Function

This is the service specific loader function for the *demoService* object. It is called once by the *cdevSystem* object in order to instantiate a single instance of the *demoService*. If more than one cdevSystem object is used by an application, the *newDemoService* function may be called more than once.

The cdevSelector Object

Each *demoService* object contains one instance of the *cdevSelector* class. This object gives the service a file descriptor that it may provide to the *cdevSystem* object for use in the *select* function. The service may then manipulate the state of the file descriptor by using the *cdevSelector* interface to indicate that one of its underlying devices requires attention.

The VERB Enumeration

This enumerated type defines a unique integer value for each of the verbs that may be applied to attributes comprising devices in this service.

The ATTR Enumeration

This enumerated type defines a unique integer identifier for each of the attributes that are comprise a device within this service.

Figure 8: demoService.h: Header file for the demoService class

```
#ifndef _DEMO_SERVICE_H_
#define _DEMO_SERVICE_H_ 1

#include <cdevService.h>
#include <cdevTranObj.h>
#include <cdevSelector.h>

// *****************************************************************
// * The demoDevice.h file defines the behavior of the demoDevice
// * objects.  These objects are crude representations of control
// * system hardware interfaces.  The structure and operation of
// * these objects are for illustration purposes only, and are
// * significantly less important than the cdev objects.
// *****************************************************************
#include <demoDevice.h>
```

Figure 8: demoService.h: (continued)

```
#define DEMO_SERVICE_NAME "demoService"

// *****************************************************************
// * Service Loader:
// * The first step in building a new service is to construct a
// * service loader function.  This function will allow cdev to
// * dynamically initialize a service by name.
// *
// * The function should adhere to this naming convention...
// *     cdevService * newXXX (char * name, cdevSystem * system);
// *     Where 'XXX' should be replaced with the name of the
// *     service class with the first character capitalized.
// *
// * The function should also be declared as 'extern "C"' to allow
// * it to be called and loaded from within an external shared
// * library module.
// *
// * This function serves only to construct a new instance of
// * the specified service, and return it to the instanciating
// * cdevSystem object.
// *****************************************************************
extern "C" cdevService *
newDemoService ( char * name, cdevSystem * system );

// *****************************************************************
// * demoService:
// * This class provides the mechanisms that the cdev system will
// * use to communicate with the demo service.
// *
// * Note that this service is inherited from the cdevService class
// * and receives much of its functionality from that class.
// *****************************************************************
class demoService : public cdevService
{
friend class demoRequestObject;
public:
    // *************************************************************
    // * Tags used by the service and its associate classes.
    // *************************************************************
    static int VALUE_TAG;
    static int STATUS_TAG;
    static int SEVERITY_TAG;
    static int UNITS_TAG;
    static int CTRLHIGH_TAG;
    static int CTRLLOW_TAG;
    static int ALRMHIGH_TAG;
    static int ALRMLOW_TAG;

    // *************************************************************
    // * demoService::VERB
    // *This is the enumerated list of verbs that are supported by
    // *devices in this service.
    // *************************************************************
    enum VERB {
        INVALID_VERB=0x00,
```

Figure 8: demoService.h: (continued)

```
        GET         =0x01, // "get"         - obtains a device value
        SET         =0x02, // "set"         - sets a device value
        VERB_MASK   =0xff
        };

    // *************************************************************
    // * demoService::ATTR
    // * This is a list of the data items that may be operated on
    // * using some or all of the demoService::VERB elements.
    // *************************************************************
    enum ATTR {
        INVALID_ATTR=0x0000,
        VAL         =0x0100, // "VAL"       - value of the device
        BDL         =0x0200, // "BDL"       - bdl value of the device
        ATTR_MASK   =0xff00
        };

    // *************************************************************
    // * demoService::demoService :
    // * This constructor is responsible for performing all service
    // * initialization.  Returns nothing.
    // *************************************************************
    demoService (char * name,
                 cdevSystem & system = cdevSystem::defaultSystem());

    // *************************************************************
    // * demoService::getFd
    // * This function will return the list of file descriptors that
    // * the demoService is using.  This will allow the file
    // * descriptors to be used in global select and poll calls
    // * performed at the cdevSystem class level.
    // *
    // * Returns CDEV_SUCCESS on success or an enumerated error.
    // *************************************************************
    int getFd ( int * &fd, int & numFd );

    // *************************************************************
    // * demoService::flush :
    // * This function flushes all communications buffers that the
    // * service may have open.
    // *
    // * Returns CDEV_SUCCESS on success or an enumerated error.
    // *************************************************************
    int flush ( void );

    // *************************************************************
    // * demoService::poll :
    // * This function polls the file descriptors used by the service
    // * until one of them becomes active or a discrete amount of
    // * time has expired.
    // *
    // * Returns CDEV_SUCCESS on success or an enumerated error.
    // *************************************************************
    int poll ( void );
```

Figure 8: demoService.h: (continued)

```
    // ************************************************************
    // * demoService::pend :
    // * Pends until the named file descriptor (or any descriptor
    // * if fd = -1) is ready.  Will pend forever if the descriptor
    // * does not become active.
    // *
    // * Returns CDEV_SUCCESS on success or an enumerated error.
    // ************************************************************
    int pend ( int fd = -1 );

    // ************************************************************
    // * demoService::pend :
    // * Pends until the named file descriptor (or any descriptor
    // * if fd = -1) is ready.  Will pend for no longer than the
    // * user specified number of seconds.
    // *
    // *Returns CDEV_SUCCESS on success or an enumerated error.
    // ************************************************************
    int pend ( double seconds, int fd = -1 );

    // ************************************************************
    // * demoService::getNameServer :
    // * This function should obtain the default name server for this
    // * object.  It does nothing for now.
    // ************************************************************
    int getNameServer(cdevDevice * &ns);

    // ************************************************************
    // * demoService::getRequestObject :
    // * This is the interface that cdev objects will use to obtain a
    // * demoRequestObject object.  The demoRequestObject represents
    // * a combined device/message pair that is associated with the
    // * demoService.
    // *
    // * Returns CDEV_SUCCESS on success or CDEV_ERROR on error.
    // ************************************************************
    int getRequestObject (char * device, char * message,
                          cdevRequestObject * &req);

protected:
    // ************************************************************
    // * demoService::~demoService :
    // * The destructor is protected to prevent it from being called
    // * directly.  The destructor performs any clean-up or shutdown
    // * operations.  Returns nothing.
    // ************************************************************
    ~demoService ( void );

    // ************************************************************
    // * demoService::enqueueTransaction :
    // * Allows the caller to place a cdevTranObj object and cdevData
    // * object into the outgoing queue.
    // ************************************************************
    int enqueueTransaction ( cdevTranObj * obj, cdevData * inbound);
```

Figure 8: demoService.h: (continued)

```
    // ************************************************************
    // * demoService::defCallback :
    // * This function is a default callback for the service.
    // ************************************************************
    static void defCallback (int, void *, cdevRequestObject &,
                             cdevData &);
private:
    // ************************************************************
    // * Queue of transaction information.  This queue is used to
    // * manage incoming requests and their associated data.  It is a
    // * simple single-linked list.  Data hiding is not violated by
    // * the public data in this class, because the entire class is a
    // * private element of the demoService class.
    // * This mini-class maintains a duplicate copy of the inbound
    // * data so that the user is free to delete it.  It also makes a
    // * copy of any pertinent context variables as they existed when
    // * the object was created.
    // ************************************************************
    class cdevTranNode
    {
    public:
        cdevTranNode * next;
        cdevTranObj  * obj;
        cdevData       inbound;
        int            VALUE_CTX;
        int            STATUS_CTX;
        int            SEVERITY_CTX;
        int            UNITS_CTX;
        int            CTRLHIGH_CTX;
        int            CTRLLOW_CTX;
        int            ALRMHIGH_CTX;
        int            ALRMLOW_CTX;

        inline cdevTranNode  ( cdevTranObj * Obj, cdevData *Inbound);
        inline ~cdevTranNode ( void ) {}
    } * transactions;

    // ************************************************************
    // * demoService::processTransaction :
    // * This function will take a cdevTranObj and will locate the
    // * demoDevice.  It will then interpret the verb and attribute
    // * and will execute the correct command on the demoDevice.
    // *
    // * Note:This object is responsible for deleting the
    // *       cdevTranObj after it has been processed.
    // ************************************************************
    int processTransaction ( demoService::cdevTranNode * node);

    // ************************************************************
    // * This is an array of pointer to the demoDevice objects that
    // * this service will manipulate.  These object are crude
    // * representations of a hardware device interface.
    // ************************************************************
    demoDevice *devices[5];
```

Figure 8: demoService.h: (continued)

```
    // ***************************************************************
    // * Default cdevCallback object for the service and the request
    // * object
    // ***************************************************************
    cdevCallback callback;

    // ***************************************************************
    // * The cdevSelector object is used to provide a file
    // * descriptor that the cdevService can use to pend for I/O
    // * events in the absence of any real I/O descriptors.
    // ***************************************************************
    cdevSelector selector;
    int          readfd[1];
};


// *******************************************************************
// * demoService::cdevTranNode::cdevTranNode :
// * This is the constructor for the cdevTranNode class.
// *******************************************************************
demoService::cdevTranNode::cdevTranNode
    ( cdevTranObj * Obj, cdevData * Inbound )
    : next(NULL), obj(Obj), VALUE_CTX(1), STATUS_CTX(0),
      SEVERITY_CTX(0), UNITS_CTX(0), CTRLHIGH_CTX(0),
      CTRLLOW_CTX(0), ALRMHIGH_CTX(0), ALRMLOW_CTX(0)
    {
    // ***************************************************************
    // * Make a duplicate copy of the inbound data
    // ***************************************************************
    if(Inbound!=NULL) inbound = *Inbound;

    // ***************************************************************
    // * Extract the pertinent context values.  Note that I am using
    // * the tag values defined in the service for speed, however, I
    // * could simply use the character string equivalents for
    // * simplicity.
    // ***************************************************************
    if(obj!=NULL)
        {
        cdevData    & context = obj->reqObj_->getContext();
        context.get(demoService::VALUE_TAG,    &VALUE_CTX);
        context.get(demoService::STATUS_TAG,   &STATUS_CTX);
        context.get(demoService::SEVERITY_TAG, &SEVERITY_CTX);
        context.get(demoService::UNITS_TAG,    &UNITS_CTX);
        context.get(demoService::CTRLHIGH_TAG, &CTRLHIGH_CTX);
        context.get(demoService::CTRLLOW_TAG,  &CTRLLOW_CTX);
        context.get(demoService::ALRMHIGH_TAG, &ALRMHIGH_CTX);
        context.get(demoService::ALRMLOW_TAG,  &ALRMLOW_CTX);
        }
    }

#endif /* _DEMO_SERVICE_H_ */
```

Figure 9: demoService.cc: Source code for the demoService class

```
#include <stdarg.h>
#include <cdevSystem.h>
#include <cdevTranObj.h>
#include <cdevClock.h>
#include <demoService.h>
#include <demoRequestObject.h>

// *****************************************************************
// * newDemoService:
// * This function is called by cdev to perform the initial
// * instanciation of a demo device service.  This function is used
// * to perform dynamic name resolution for the services that are
// * specified within the DDL file or the name server.
// *****************************************************************
cdevService * newDemoService ( char * name, cdevSystem * system )
    {
    return new demoService(name, *system);
    }

// *****************************************************************
// * demoService::demoService :
// * This constructor is responsible for performing all service
// * initialization.  Returns nothing.
// *****************************************************************
demoService::demoService ( char * name, cdevSystem & system)
    : cdevService (name, system), transactions(NULL),
      callback    (defCallback, NULL)
    {
    // ************************************************************
    // * Obtain the values of the static tags used by this service
    // ************************************************************
    cdevData::tagC2I("value",       &VALUE_TAG);
    cdevData::tagC2I("status",      &STATUS_TAG);
    cdevData::tagC2I("severity",    &SEVERITY_TAG);
    cdevData::tagC2I("units",       &UNITS_TAG);
    cdevData::tagC2I("controlHigh", &CTRLHIGH_TAG);
    cdevData::tagC2I("controlLow",  &CTRLLOW_TAG);
    cdevData::tagC2I("alarmHigh",   &ALRMHIGH_TAG);
    cdevData::tagC2I("alarmLow",    &ALRMLOW_TAG);

    // ************************************************************
    // * At this point, I will transfer pointers to the external
    // * demoDevice objects into my array of demoDevice pointers.
    // *************************************************************
    devices[0] = &DEVICE0;
    devices[1] = &DEVICE1;
    devices[2] = &DEVICE2;
    devices[3] = &DEVICE3;
    devices[4] = &DEVICE4;
    }

// *****************************************************************
// * demoService::~demoService :
// * The destructor is protected to prevent it from being called
// * directly.  The destructor performs any clean-up or shutdown
```

Figure 9: demoService.cc: (continued)

```
// * operations.  Returns nothing.
// *****************************************************************
demoService::~demoService ( void )
    {
    // ************************************************************
    // * Delete all outstanding transaction object nodes.  Note that
    // * typically a transaction object should not be deleted until
    // * after it has been processed, because a group object may be
    // * waiting on it. However, the removeFromGrps method will
    // * remove the transaction object from any groups that may be
    // * waiting for it.
    // ************************************************************
    while(transactions != NULL)
        {
        cdevTranNode * node = transactions;
        transactions        = node->next;
        if(node->obj!=NULL)
            {
            if(node->obj->status()!=-1) node->obj->removeFromGrps();
            delete node->obj;
            }
        delete node;
        }
    }

// *****************************************************************
// * demoService::getFd
// * This function will return the list of file descriptors that the
// * demoService is using.  This will allow the file descriptors to
// * be used in global select and poll calls performed at the
// * cdevSystem class level.
// *
// * Returns CDEV_SUCCESS on success or CDEV_ERROR on error.
// *****************************************************************
int demoService::getFd ( int * &fd, int & numFd )
    {
    readfd[0] = selector.readfd();
    fd        = readfd;
    numFd     = 1;
    return CDEV_SUCCESS;
    }


// *****************************************************************
// * demoService::flush :
// * This function flushes all communications buffers that the
// * service may have open.
// * Returns CDEV_SUCCESS on success or an enumerated error.
// * Note: This service does not use communications buffers.
// *****************************************************************
int demoService::flush ( void )
    {
    return CDEV_SUCCESS;
    }
```

Figure 9: demoService.cc: (continued)

```
// *****************************************************************
// * demoService::poll :
// * This function submits a pend request that is short enough to
// * allow only one cdevTranObj to be dequeued and submitted to the
// * demoDevice for processing.
// * Returns CDEV_SUCCESS on success or an enumerated error.
// *****************************************************************
int demoService::poll ( void )
    {
    return pend(0.0001);
    }

// *****************************************************************
// * demoService::pend :
// * This function will process all of the cdevTranObj messages that
// * are residing in the queue.
// * Returns CDEV_SUCCESS on success or an enumerated error.
// *****************************************************************
int demoService::pend ( int )
    {
    while(transactions != NULL)
        {
        cdevTranNode * node = transactions;
        transactions = node->next;
        processTransaction (node);
        }
    selector.purge();
    return CDEV_SUCCESS;
    }

// *****************************************************************
// * demoService::pend :
// * Pends until the named file descriptor (or any file descriptor
// * if fd = -1) is ready.  Will pend for no longer than the user
// * specified number of seconds.
// * Returns CDEV_SUCCESS on success or CDEV_ERROR on error.
// *****************************************************************
int demoService::pend ( double seconds, int )
    {
    cdevTimeValue t(seconds);
    cdevClock timer;

    timer.schedule(NULL, t);

    while(transactions!=NULL && !timer.expired())
        {
        cdevTranNode * node = transactions;
        transactions        = node->next;
        processTransaction(node);
        selector.removeEvent();
        }

    if(transactions==NULL) selector.purge();
    return CDEV_SUCCESS;
    }
```

Figure 9: demoService.cc: (continued)

```
// *****************************************************************
// * demoService::getRequestObject :
// * This is the interface that cdev objects will use to obtain a
// * demoRequestObject object.  The demoRequestObject represents a
// * combined device and message pair that is associated with the
// * demoService.
// *
// * Returns CDEV_SUCCESS on success or CDEV_ERROR on error.
// *****************************************************************
int demoService::getRequestObject ( char * device, char * message,
                                    cdevRequestObject * &req)
    {
    req = new demoRequestObject (device, message, system_);
    return (req ? CDEV_SUCCESS : CDEV_ERROR);
    }

// *****************************************************************
// * demoService::enqueueTransaction :
// * Allows the caller to place a cdevTranObj object and cdevData
// * object into the outgoing queue.  This function also calls the
// * selector.insertEvent() method to activate the file descriptor.
// *****************************************************************
int demoService::enqueueTransaction ( cdevTranObj * obj,
                                      cdevData * inbound )
    {
    int result = CDEV_SUCCESS;

    if(obj!=NULL)
        {
        cdevTranNode * node = transactions, * prev = NULL;
        while(node!=NULL)
            {
            prev = node;
            node = prev->next;
            }
        node = new cdevTranNode(obj, inbound);
        if(prev==NULL) transactions = node;
        else           prev->next   = node;
        selector.insertEvent();
        }
    else result = CDEV_INVALIDARG;

    return result;
    }

// *****************************************************************
// * demoService::processTransaction :
// * This function will take a cdevTranNode and will locate the
// * device associated with it.  It will then interpret the verb and
// * attribute and will execute the correct command on the device.
// *
// * Note:This object is responsible for deleting the cdevTranNode
// * after it has been processed.
// *****************************************************************
int demoService::processTransaction ( cdevTranNode * node )
```

Figure 9: demoService.cc: (continued)

```
{
cdevTranObj       * obj    = node->obj;
demoRequestObject * reqObj =
   obj?(demoRequestObject *)obj->reqObj_:NULL;
cdevData          * out    =
   obj?obj->resultData_:NULL;
char              * name   =
   obj?(char *)obj->reqObj_->device().name():NULL;
int                 result = CDEV_SUCCESS;
int                 idx    = 0;
cdevData            data;

// ***********************************************************
// * Ensure that if the obj is valid it has a valid cdevData
// * object for its result.
// ***********************************************************
if(obj!=NULL && obj->resultData_==NULL) obj->resultData_=&data;
out = obj?obj->resultData_:NULL;

// ***********************************************************
// * Ensure that a valid transaction object was submitted.
// * Remembering to produce an error that the user can catch.
// ***********************************************************
if(obj==NULL) result=
   reqObj->emitError(CDEV_INVALIDARG, "service");

// ***********************************************************
// * Test to ensure that the device name specified is valid and
// * obtain the index to the demoDevice in the array.  A better
// * implementation might cache a pointer to the specific device
// * within the demoRequestObject.
// ***********************************************************
else if(strncmp(reqObj->device().name(), "DEVICE", 6)!=0 ||
    (idx = atoi(reqObj->device().name()+6))<0  || idx > 9)
   {
   result = reqObj->emitError(CDEV_INVALIDSVC, "service");
   }

// ***********************************************************
// * Process the specific message.
// ***********************************************************
else if(result==CDEV_SUCCESS)
   {
   int attr = -1;

   // *********************************************************
   // * Obtain the index of the attribute within the demoDevice
   // *********************************************************
   if((reqObj->command() & ATTR_MASK)==VAL)
      attr=devices[idx]->getIndex("VAL");

   else if((reqObj->command() & ATTR_MASK)==BDL)
      attr=devices[idx]->getIndex("BDL");

   switch(reqObj->command())
```

Figure 9: demoService.cc: (continued)

```
          {
          // ****************************************************
          // * "get VAL" || "get BDL" messages
          // * This will populate the outbound cdevData object
          // *with all of the context items that were in
          // *place when the transaction object was created.
          // ****************************************************
          case GET         | VAL:
          case GET         | BDL:
          if(node->VALUE_CTX)
              out->insert(VALUE_TAG,
                          devices[idx]->getValue(attr));
          if(node->STATUS_CTX)
              out->insert(STATUS_TAG,
                          devices[idx]->getStatus(attr));
          if(node->SEVERITY_CTX)
              out->insert(SEVERITY_TAG,
                          devices[idx]->getSeverity(attr));
          if(node->UNITS_CTX)
              out->insert(UNITS_TAG,
                          devices[idx]->getUnits(attr));
          if(node->CTRLLOW_CTX)
              out->insert(CTRLLOW_TAG,
                          devices[idx]->getControlLow(attr));
          if(node->CTRLHIGH_CTX)
              out->insert(CTRLHIGH_TAG,
                          devices[idx]->getControlHigh(attr));
          if(node->ALRMLOW_CTX)
              out->insert(ALRMLOW_TAG,
                          devices[idx]->getAlarmLow(attr));
          if(node->ALRMHIGH_CTX)
              out->insert(ALRMHIGH_TAG,
                          devices[idx]->getAlarmHigh(attr));
          break;

          // ****************************************************
          // * "set VAL" || "set BDL" message.  These messages
          // * will allow the caller to set the current value of
          // * the device.  This section ensures that an error
          // * is emitted if the operation fails.
          // ****************************************************
          case SET         | VAL:
          if(node->inbound.getType(VALUE_TAG)==CDEV_DOUBLE)
              {
              double val;
              size_t elems = 0;

              node->inbound.getElems(VALUE_TAG, &elems);
              if(elems==1 &&
                 node->inbound.get(VALUE_TAG, &val)==CDEV_SUCCESS)
                 {
                 if(devices[idx]->setValue(attr, val)==0)
                     {
                     result = CDEV_SUCCESS;
                     }
```

Figure 9: demoService.cc: (continued)

```
                else
                    {
                    system_.reportError
                        (2, demoService::name(), *reqObj,
                        "Value out of range for this device\n",);
                    result = CDEV_INVALIDARG;
                    }
                }
            else result =
                reqObj->emitError(CDEV_CONFLICT, "service");
            }
        else result =
            reqObj->emitError(CDEV_INVALIDARG, "service");
        break;

        // *****************************************************
        // * Invalid combination of verb and attribute
        // *****************************************************
        default:
        result = reqObj->emitError(CDEV_ERROR, "service");
        break;
        }

    // *********************************************************
    // * Fire the user specified callback function to give the
    // * user the result.
    // *********************************************************
    if(obj->userCallback_!=NULL)
        {
        (*(obj->userCallback_->callbackFunction()))
            (result==CDEV_SUCCESS?CDEV_SUCCESS:CDEV_ERROR,
             obj->userCallback_->userarg(),
             *reqObj,
             *out);
        }
    }

// ************************************************************
// * Delete the cdevTranNode object.
// ************************************************************
if(node!=NULL)
    {
    if(node->obj!=NULL)
        {
        node->obj->removeFromGrps();
        delete node->obj;
        }
    delete node;
    }

// ************************************************************
// * Return the result.
// ************************************************************
return result;
}
```

Figure 9: demoService.cc: (continued)

```
// *****************************************************************
// * demoService::defCallback :
// * This function is a default callback function for the service.
// *****************************************************************
void demoService::defCallback ( int, void *, cdevRequestObject &,
                                cdevData &)
    {
    }
```

The demoRequestObject Object

The *demoRequestObject* C++ class is derived from the *cdevRequestObject* class. When the caller uses the *cdevDevice* object to submit a supported message to one of the *demoDevices*, a *demoRequestObject* is created. This object defines the *send, sendNoBlock* and *sendCallback* methods that are used to submit messages to the *demoService* object. This request object restricts itself to using the *enqueueTransaction* method of the *demoService* object to submit messages. A real implementation might precalculate and cache many of the data variables and pointers within the request object. These variable can be retrieved by the service during successive executions of the message.

Figure 10: demoRequestObject.h: Header file for the demoRequestObject class

```
#ifndef _DEMO_REQUEST_OBJECT_H_
#define _DEMO_REQUEST_OBJECT_H_

#include <cdevTranObj.h>
#include <cdevGroup.h>
#include <cdevErrCode.h>
#include <demoService.h>

// *****************************************************************
// * class demoRequestObject:
// * The demoRequestObject class provides the interface for sending
// * messages to a demoDevice.  All device/message commands are
// * routed
// * through a demoRequestObject either directly or indirectly.
// *
// * Note that this object is inherited from the standard
// * cdevRequestObject and derives most of its functionality from
// * that class.
// *****************************************************************
class demoRequestObject : public cdevRequestObject
{
public:
    // ************************************************************
    // * demoRequestObject::demoRequestObject :
    // * This constructor initializes a device/message combination
    // * associated with a demoDevice.  Returns nothing.
    // ************************************************************
    demoRequestObject ( char * device, char * message,
        cdevSystem & system=cdevSystem::defaultSystem());
```

Figure 10: demoRequestObject.h: (continued)

```
// ***************************************************************
// * demoRequestObject::~demoRequestObject :
// * This destructor performs any deallocation operations
// * necessary, prior to the destruction of the object.
// * Returns nothing.
// ***************************************************************
~demoRequestObject ( void ) {}

// ***************************************************************
// * demoRequestObject::send :
// * The send interface is used to provide synchronous I/O with
// * the service.
// *
// * Returns CDEV_SUCCESS on success or an enumerated error.
// ***************************************************************
int send ( cdevData & in, cdevData & out )
    { return send(&in, &out); }
int send ( cdevData * in, cdevData & out )
    { return send(in, &out);  }
int send ( cdevData & in, cdevData * out )
    { return send(&in, out);  }
int send ( cdevData * in, cdevData * out );

// ***************************************************************
// * demoRequestObject::sendNoBlock :
// * The sendNoBlock interface is used in concert with cdevGroup
// * or cdevSystem to execute a series of operations.
// *
// * Returns CDEV_SUCCESS on success or an enumerated error.
// ***************************************************************
int sendNoBlock (cdevData & in, cdevData & out)
    { return sendNoBlock(&in, &out); }
int sendNoBlock (cdevData * in, cdevData & out)
    { return sendNoBlock(in, &out); }
int sendNoBlock (cdevData & in, cdevData * out)
    { return sendNoBlock(&in, out); }
int sendNoBlock (cdevData * in, cdevData * out);

// ***************************************************************
// * demoRequestObject::sendCallback :
// * The sendCallback interface provides asynch communications
// * with the service.
// *
// *Returns CDEV_SUCCESS on success or an enumerated error.
// ***************************************************************
int sendCallback (cdevData & in, cdevCallback & callback)
    { return sendCallback(&in, callback); }
int sendCallback (cdevData * in, cdevCallback & callback);

// ***************************************************************
// * demoRequestObject::emitError :
// * This method is used to emit a descriptive error if a
// * failure occurs while transmitting or receiving a message.
// ***************************************************************
int emitError ( int result, char * function );
```

Figure 10: demoRequestObject.h: (continued)

```
    // ***********************************************************
    // * demoRequestObject::className :
    // * This function returns the name of the class as a constant
    // * string
    // ***********************************************************
    const char * className ( void ) const
        {
        return "demoRequestObject";
        }

    // ***********************************************************
    // * demoRequestObject::command :
    // * Returns the combined verb and attr that describe this
    // * demoRequestObject.  This command is used by the demoService
    // * to determine the operation that is specified by this object.
    // ***********************************************************
    int command ( void ) { return (verb | attr); }

private:
    demoService::VERB verb;
    demoService::ATTR attr;
    };

#endif /* _DEMO_REQUEST_OBJECT_H_ */
```

Figure 11: demoRequestObject.cc: Source for the demoRequestObject class

```
#include <ctype.h>
#include <demoRequestObject.h>

// ***************************************************************
// * demoRequestObject::demoRequestObject :
// * This constructor initializes the internals of a device/message
// * pair associated with the compound device.
// *
// * Returns nothing.
// ***************************************************************
demoRequestObject::demoRequestObject
    ( char *device, char *message, cdevSystem & system)
    : cdevRequestObject(device, message, system),
      verb(demoService::INVALID_VERB),
      attr(demoService::INVALID_ATTR)
    {
    char Verb[32];
    char Attr[32];
    char * mptr = message;
    char * vptr = Verb;
    char * aptr = Attr;
```

Figure 11: demoRequestObject.cc: (continued)

```
    // ***************************************************************
    // * Extract the verb and the attribute from the message string
    // * that was provided by the caller.
    // ***************************************************************
    while(*mptr &&  isspace(*mptr)) mptr++;
    while(*mptr && !isspace(*mptr)) *(vptr++) = *(mptr++);
    *vptr = 0;
    while(*mptr &&  isspace(*mptr)) mptr++;
    while(*mptr && !isspace(*mptr)) *(aptr++) = *(mptr++);
    *aptr = 0;

    // ***************************************************************
    // * Evaluate the Verb element and assign the appropriate value
    // * to the enumerated verb variable.
    // ***************************************************************
    if     (!strcmp(Verb, "get"))        verb = demoService::GET;
    else if(!strcmp(Verb, "set"))        verb = demoService::SET;

    // ***************************************************************
    // * Evaluate the Attr element and assign the appropriate value
    // * to the enumerated attr variable.
    // ***************************************************************
    if     (!strcmp(Attr, "VAL"))        attr = demoService::VAL;
    else if(!strcmp(Attr, "BDL"))        attr = demoService::BDL;
    }

// *******************************************************************
// * demoRequestObject::sendNoBlock :
// * This function allows the caller to submit an asynchronous
// * message to the server for processing.
// *******************************************************************
int demoRequestObject::sendNoBlock (cdevData * in, cdevData * out)
    {
    // ***************************************************************
    // * Get a pointer to the demoService to handle this request
    // ***************************************************************
    demoService   * svc  = (demoService *)service_;

    // ***************************************************************
    // * Construct a cdev transaction object to track this
    // * transaction. Use the disableDeleteCbk method to prevent the
    // * cdevCallback object from being deleted by the cdevTranObj.
    // ***************************************************************
    cdevTranObj   * xobj =
        new cdevTranObj(&system_, this, out, &svc->callback);
    xobj->disableDeleteCbk();

    // ***************************************************************
    // * Enqueue the transaction object so that it may be processed
    // * later.  The result returned by the enqueueTransaction
    // * function is returned by this function.
    // ***************************************************************
    return emitError
        (svc->enqueueTransaction(xobj, in), "sendNoBlock");
    }
```

Figure 11: demoRequestObject.cc: (continued)

```
// ****************************************************************
// * demoRequestObject::sendCallback :
// * This function allows the caller to submit an asynchronous
// * message to the server for processing.
// ****************************************************************
int demoRequestObject::sendCallback
    (cdevData * in, cdevCallback & callback)
    {
    // ************************************************************
    // * Get a pointer to the demoService to handle this request
    // ************************************************************
    demoService    * svc  = (demoService *)service_;

    // ************************************************************
    // * Make a cdev transaction object to track this transaction.
    // * Note that the user specified callback object is provided
    // * to the object and that no cdevData object is provided for
    // * the result.
    // ************************************************************
    cdevTranObj  * xobj =
        new cdevTranObj(&system_, this, NULL, &callback);
    xobj->disableDeleteCbk();

    // ************************************************************
    // * Enqueue the transaction object so that it may be processed
    // * later.  The result returned by the enqueueTransaction
    // * function is returned by this function.
    // ************************************************************
    return emitError
        (svc->enqueueTransaction(xobj, in), "sendCallback");
    }

// ****************************************************************
// * demoRequestObject::send :
// *The send interface is used for synchronous I/O with the service.
// *Returns CDEV_SUCCESS on success or CDEV_ERROR on error.
// *Note:Do not delete the transaction object, it will be deleted by
// * the processTransaction method in the service.
// ****************************************************************
int demoRequestObject::send ( cdevData * in, cdevData * out )
    {
    int           result = CDEV_SUCCESS;
    cdevTranObj * xobj;

    // *************************************************************
    // * Get a pointer to the demoService to handle this request
    // *************************************************************
    demoService * svc = (demoService *)service_;

    // *************************************************************
    // * Construct a group object that can be used to pend for the
    // * completion of the send event.
    // *************************************************************
    cdevGroup group(DEFAULT_BLOCK_SIZE, system());
```

Figure 11: demoRequestObject.cc: (continued)

```
    // ***************************************************************
    // * Start the group prior to creating the transaction object.
    // ***************************************************************
    group.start();

    // ***************************************************************
    // * Construct a cdev transaction object to track this
    // * transaction. Note that the callback object that exists
    // * within the service is specified as the callback object for
    // * this transaction.
    // ***************************************************************
    xobj = new cdevTranObj(&system_, this, out, &svc->callback);
    xobj->disableDeleteCbk();

    // ***************************************************************
    // * End the group and prepare to pend for the completion of the
    // * task.
    // ***************************************************************
    group.end();

    // ***************************************************************
    // * Enqueue the transaction object so that it may be processed
    // * later.
    // ***************************************************************
    if((result=svc->enqueueTransaction(xobj, in))==CDEV_SUCCESS)
        {
        // ***********************************************************
        // * Pend for a period of not more than 5 seconds, or until
        // * the message has been completed.
        // ***********************************************************
        result = group.pend(5.0);
        }

    // ***************************************************************
    // * Use the reportError function to report an error if one
    // * occurred.
    // ***************************************************************
    return emitError(result, "send");
    }

// *******************************************************************
// * demoRequestObject::emitError :
// * This method is used to emit a descriptive error if a failure
// * occurs while transmitting or receiving a message.
// *******************************************************************
int demoRequestObject::emitError ( int result, char * function )
    {
    switch(result)
        {
        case CDEV_SUCCESS:
        system().reportError(CDEV_SEVERITY_INFO, service().name(),
            *this, "%s operation successful\n", function);
        break;
```

Figure 11: demoRequestObject.cc: (continued)

```
case CDEV_INVALIDOBJ:
system().reportError(CDEV_SEVERITY_ERROR, service().name(),
   *this, "Invalid object detecting in %s operation\n",
   function);
break;

case CDEV_INVALIDARG:
system().reportError(CDEV_SEVERITY_ERROR, service().name(),
   *this,
   "Invalid argument for \"%s\" message in %s operation\n",
   message(), function);
break;

case CDEV_INVALIDSVC:
system().reportError(CDEV_SEVERITY_SEVERE, service().name(),
   *this, "Wrong service loaded during dynamic loading\n");
break;

case CDEV_NOTCONNECTED:
system().reportError(CDEV_SEVERITY_ERROR, service().name(),
   *this, "Network connection error in %s operation\n",
   function);
break;

case CDEV_IOFAILED:
system().reportError(CDEV_SEVERITY_ERROR, service().name(),
   *this, "I/O failure during %s operation\n", function);
break;

case CDEV_CONFLICT:
system().reportError(CDEV_SEVERITY_WARN, service().name(),
*this,
"Data type provided is inconsistent with \"%s\" message\n",
message());
break;

case CDEV_NOTFOUND:
system().reportError(CDEV_SEVERITY_WARN, service().name(),
*this,
"Tagged data item missing from cdevData in %s operation\n",
function);
break;

case CDEV_TIMEOUT:
system().reportError(CDEV_SEVERITY_WARN, service().name(),
   *this,
   "Timed-out while waiting for response in %s operation\n",
   function);
break;

case CDEV_CONVERT:
system().reportError(CDEV_SEVERITY_WARN, service().name(),
   *this, "Data conversion error in %s operation\n",
   function);
break;
```

Figure 11: demoRequestObject.cc: (continued)

```
        case CDEV_DISCONNECTED:
        system().reportError(CDEV_SEVERITY_WARN, service().name(),
            *this, "Disconnected from service during %s operation\n",
            function);
        break;

        case CDEV_RECONNECTED:
        system().reportError(CDEV_SEVERITY_INFO, service().name(),
            *this, "Reconnected to service during %s operation\n",
            function);
        break;
        }
    return result;
    }
```

Makefile for the demoService

The following *Makefile* can be used to build the shared object for the *demoService*. This makefile uses the *Makefile.common* of the cdev system for its base. It has the additional requirement that the *CDEVSHOBJ* environment variable must be defined. This variable defines where the shared objects used by cdev should be stored.

Figure 12: Makefile for the demoService

```
####################################################################
# Makefile for demoService shared object
####################################################################
CDEVROOT = $(CDEV)
SHOBJ=YES
include $(CDEVROOT)/src/Makefile.common

LIBS = -L$(LIBDIR) -lcdev -ly -ll -lm
OBJS = demoService.o demoRequestObject.o demoDevice.o
CLASS_INCLUDES = -I.
CXXEXTRA = -Aa +a1 -z +z -pta -g $(CLASS_INCLUDES)

all: demoService.so

demoService.so : $(OBJS)
    @echo Creating demoService shared object
    @rm -f demoService.so
    CC -b -o demoService.so $(OBJS)
    @cp demoService.so $(CDEVSHOBJ)/demoService.so
    @echo done

clean:
    @echo Cleaning up source directory
    @rm -rf *.so *.a *~ *.o core ptrepository TC.Cache
    @echo done
```

Device Definition File for the demoService

The following *Device Definition Language (DDL)* file defines the verbs and attributes supported by the demoDevice. It them defines the instances of demo devices that are available for use. This DDL file may be used in conjunction with the *cdevUtil* application to communicate with the *demoDevice* control system.

Figure 13: Device Definition Language file for the demoService

```
/*
 * Service definitions
 */
service demo {
    tags {XYZ}
    }

/*
 * Class definitions
 */
class demoDevice {
    verbs { get, set}
    attributes
        {
        VAL demo {};
        BDL demo {};
        }
    }

/*
 * Device instanciations
 */
demoDevice :
    DEVICE0
    DEVICE1
    DEVICE2
    DEVICE3
    DEVICE4;
```