CDEV Reference Guide

Chip Watson, Jie Chen, Danjin Wu, Walt Akers

Version 1.5 - December 9, 1996

TJNAF - Thomas Jefferson National Accelerator Facility

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

TRADEMARKS:

SURA/CEBAF:

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

DISCLAIMER:

Table of Contents

List of Figures

1. The cdevCallback Class

Overview of the cdevCallback Class

The *cdevCallback* C++ class is used by the sendCallback method to specify the callback function that should be executed when the message has been processed. This class also allows the caller to include a user specified argument that will be provided as a parameter in the callback function when it is executed.

Figure 1: Object model of the cdevCallback class

```
                        class cdevCallback
methods:
   callbackFunction (void)             : cdevCallbackFunction
   userarg          (void)             : void *
   operator ==      (cdevCallback &)   : int
   operator !=      (cdevCallback &)   : int
   fireCallback     (...)              : void
   isTransactionDone(void)    {static}: int
```

Public Methods of the cdevCallback Class

cdevCallback

cdevCallback (void);
*cdevCallback (cdevCallbackFunction func, void * userarg);*
cdevCallback (cdevCallback & callback);

This is the constructor for the callback object. It sets the internal pointers to the user specified callback function and userarg pointer.

callbackFunction

cdevCallbackFunction callbackFunction (void);

Obtains a pointer to the callback function that is stored within the cdevCallback object.

userarg

*void * userarg (void);*

Obtains a pointer to the user argument that is stored within the cdevCallback object.

operator ==

int operator == (const cdevCallback& callback);

Used to compare two cdevCallback objects. Two objects are considered equal if they point to the same callback function and user argument.

operator !=

int operator != (const cdevCallback& callback);

Used to compare two cdevCallback objects. Two objects are considered equal if they point to the same callback function and user argument.

fireCallback

void fireCallback (int status,
*void * userarg,*
cdevRequestObject & req,
cdevData & result,
int partialTransaction = 0);

This method allows the caller to dispatch the callback function with the specified parameters. If the callback represents a response that is one of many that will occur, then the partialTransaction flag should be set to non-zero. The value of this flag can be retrieved by calling the static cdevCallback::isTransactionDone method.

isTransactionDone

int isTransactionDone (void);

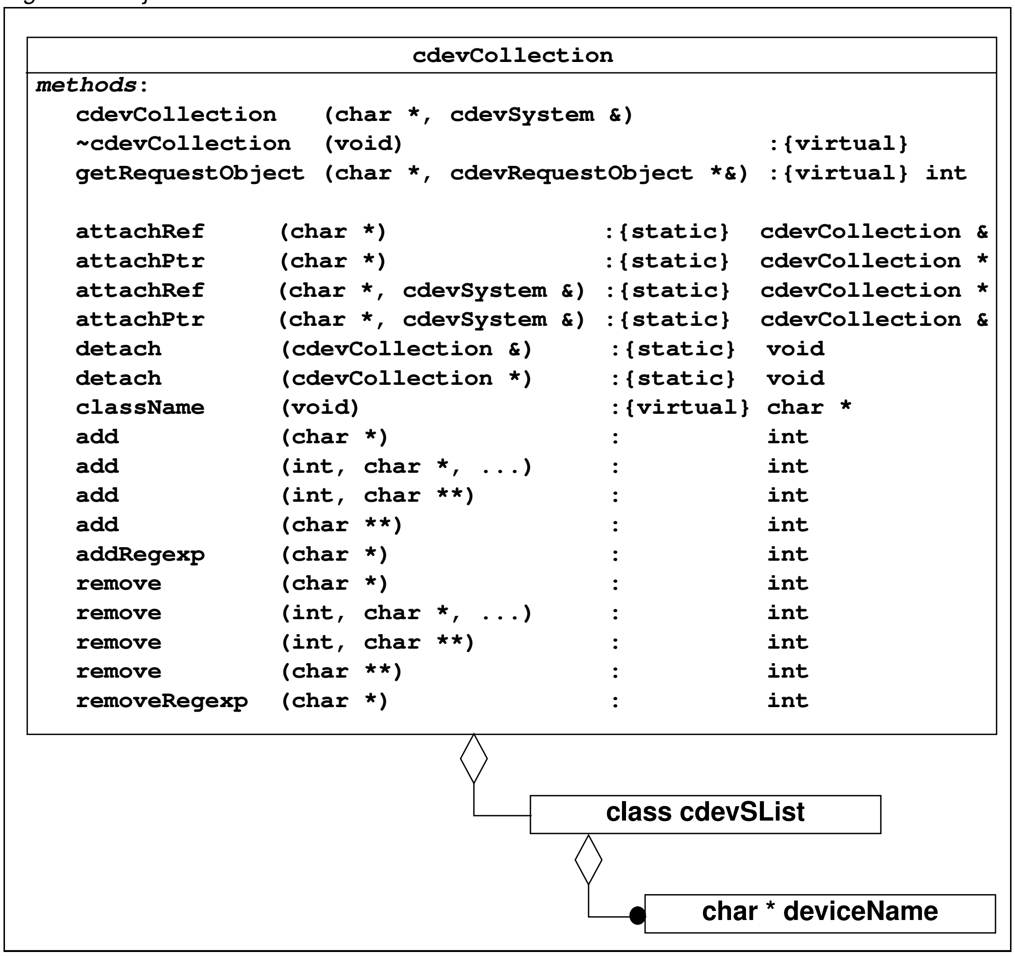
This method returns an integer indicating if this callback indicates the end of the transaction. In the event of multi-part transactions such as “monitorOn”, this method will return a non-zero value indicating that more callbacks can be expected. In the event of monotonic operations such as “get” or “set”, or when the last monitor callback is received, this method will return zero.

2. The cdevCollection Class

Overview of the cdevCollection Class

The cdevCollection class provides a mechanism for sending a message to a collection of named devices. The class maintains a list of CDEV devices that are associated through an entry in the CDEV DDL file. The developer also dynamically create a cdevCollection device.

Figure 2: Object Model of the cdevCollection Class



Public Methods of the cdevCollection Class

cdevCollection

*cdevCollection (char * name, cdevSystem & system);*

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

- This method is ***protected*** to prevent *cdevCollections* from being instantiated directly. New *cdevCollection* instances are created by using the *attachPtr* or *attachRef* method of the *cdevCollection* class.
- The name that is provided to the *cdevCollection* is the device name that CDEV will use to refer to it. The name must be unique within the CDEV system of objects and cannot be the same as the name of any regular *cdevDevice* that exists in the CDEV DDL file.

- The *cdevSystem* reference that is provided is the *cdevSystem* instance that will be used to *poll*, *pend* and *flush* the *cdevCollection* object.
- If the name provided is the name of a *cdevCollection* that has been specified in the *CDEV DDL* file, then the list of devices will be populated from the names that are specified. Otherwise, an empty *cdevCollection* instance will be created.

~cdevCollection *virtual ~cdevCollection (void);*

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

- This method is protected to prevent the *cdevCollection* object from being destroyed by the application. This method should only be called by the *cdevSystem* object when the application is terminating.
- Because the *cdevCollection* object will normally be referred to as a *cdevDevice* object, this destructor is virtual to ensure that the ***'most senior'*** destructor is called first.
- This method will delete all device name strings that are associated with the *cdevCollection* object.

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

This method is called in order to obtain the cdevRequestObject associated with the specified device/msg combination. This method returns CDEV_SUCCESS if the cdevRequestObject was successfully obtained. This method has the following properties.

- The *cdevRequestObject* pointer that is returned is actually a pointer to a *cdevCollectionRequest* object.
- If a *cdevCollectionRequest* already exists for the specified device/msg combination, it will be returned. Otherwise, the method will call the *attachPtr* method of the *cdevCollectionRequest* in order to obtain a new request object.

attachRef/attachPtr

*cdevCollection & attachRef(char * name);*
*cdevCollection & attachRef(char * name, cdevSystem & sys);*
*cdevCollection * attachPtr(char * name);*
*cdevCollection * attachPtr(char * name, cdevSystem &sys);*

These methods are used to obtain a pointer or reference to a cdevCollection object. The name variable specifies the name of the collection to be attached. These method have the following properties.

- All functionality for these methods is stored in the *attachPtr (char *, cdevSystem &)* method. All other methods will call this one.

- The method will first determine if the device has already been created. If it has been created and its *className* method returns “*cdevCollection*”, then the existing *cdevCollection* will be returned. If a non-collection *cdevDevice* of the same name has already been constructed, then NULL will be returned.

- If a device with the specified name has not already been created, then the method will search the *cdevDirectory* for the specified name. If the device name is in the *cdevDirectory* AND is of class “*collection*”, then a new *cdevCollection* will be instantiated and returned. If a non-collection *cdevDevice* of the same name exists in the *cdevDirectory*, then NULL will be returned.

- If a device with the specified name has not been instantiated and does not exist in the *cdevDirectory*, then a new, empty *cdevCollection* will be created with the specified name.

- If NULL is returned to one of the *attachRef* methods, then a reference to a *cdevErrorCollection* object will be returned to the caller.

- The *refCount* property of the *cdevCollection* will be incremented each time a copy of the *cdevCollection* object is obtained by the application.

detach

int detach (cdevCollection & col);
*int detach (cdevCollection * col);*

This method is called to detach a pointer or reference to a cdevCollection object. It has the following properties.

- This method decrements the *refCount* property of the *cdevCollection* object.

- When the *refCount* property is decremented to 0, then the *cdevCollection* object will be deleted.

className

*char * className (void);*

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

add

*int add (char *name);*

This method will add the individual name to the list of device names that are in the cdevCollection. This method has the following properties.

- If a *cdevCollectionRequest* object has been instantiated for this *cdevCollection* device, then the operation will fail and **CDEV_INVALIDOP** will be returned.
- If the name provided is NULL or empty, then the operation will fail and **CDEV_INVALIDOP** will be returned.
- If the name specified already exists in the collection of names, then the operation will fail and **CDEV_WARNING** will be returned.
- If the name specified is not already in the list, then the method will append the device name to the end of the list and return **CDEV_SUCCESS**.

add *int add (int num, char *firstname,...);*

This method will attempt to add a list of num names to the cdevCollection. This method has the following properties.

- If a *cdevCollectionRequest* object has been instantiated for this *cdevCollection* device, then the operation will fail and **CDEV_INVALIDOP** will be returned.
- This method will call the *add (char *name)* method for each name that is specified.
- If the add method fails to add any of the device name strings to the list, then the error code that was generated by the call to *add (char * name)* will be returned.
- If an error occurs while adding a name to the list, the method will continue with the next name until the last name is reached.
- If no errors occur while adding names to the list, then **CDEV_SUCCESS** will be returned to the caller.

add *int add (int num, char **names);*

This method will attempt to add *num* device names from the *names* array to the list using the *add (char *name)* method of the *cdevCollection*. This method has the same properties as the *add (int num, char * firstname,...)* method.

add *int add (char **names);*

This method will attempt to add each device name from the NULL terminated *names* array to the list using the *add (char *name)* method of the *cdevCollection*. This method has the same properties as the *add (int num, char * firstname,...)* method.

addRegexp *int addRegexp (char *regexp);*

This method will query the system name server for the list of all names in the *cdevDirectory* that match the specified regular expression. the array of names will be added to the list using the *add (int num, char ** names)* method and the result of this operation will be returned.

remove *int remove (char *name);*

This method will remove the specified device name from list of device names that is stored in the *cdevCollection* object. This method has the following properties.

- If a *cdevCollectionRequest* object has been instantiated for this *cdevCollection* device, then the operation will fail and **CDEV_INVALIDOP** will be returned.
- If the name provided is NULL or empty, then the operation will fail and **CDEV_WARNING** will be returned.
- If the name specified does not exist in the collection of names, then the operation will fail and **CDEV_WARNING** will be returned.
- If the name specified exists in the list, then it will be removed and **CDEV_SUCCESS** will be returned.

remove *int remove (int num, char *name,...);*

This method will attempt to remove a list of num names from the *cdevCollection*. This method has the following properties.

- If a *cdevCollectionRequest* object has been instantiated for this *cdevCollection* device, then the operation will fail and **CDEV_INVALIDOP** will be returned.
- This method will call the *remove (char *name)* method for each name that is specified.
- If the remove method fails to remove any of the device name strings from the list, then the error code that was generated by the call to *remove (char * name)* will be returned.
- If an error occurs while removing a name from the list, the method will continue with the next name until the last name is reached.
- If no errors occur while removing names from the list, then **CDEV_SUCCESS** will be returned to the caller.

remove *int remove (int num, char **names);*

This method will attempt to remove *num* device names as specified in the *names* array from the list using the *remove (char *name)* method of the *cdevCollection*. This method has the same properties as the *remove (int num, char * name,...)* method.

remove *int remove (char **names);*

This method will attempt to remove each device name as specified in the NULL terminated *names* array from the list using the *remove (char *name)* method of the *cdevCollection*. This method has the same properties as the *remove (int num, char * name,...)* method.

removeRegexp *int removeRegexp (char *regexp);*

This method will query the system name server for the list of all names in the *cdevDirectory* that match the specified regular expression. the array of names will be removed from the list using the *remove (int num, char ** names)* method and the result of this operation will be returned.

3. 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.

Figure 3: Object Model of the cdevCollectionRequest Class

```
                         cdevCollectionRequest
attributes:
   RESULT_CODE_TAG: int

methods:
   cdevCollectionRequest    (char **, int, char *, cdevSystem &)
   ~cdevCollectionRequest   (void)  :{virtual}

   attachPtr              (...)   :{static} cdevCollectionRequest *
   className                (void)  :{virtual} char *
   resultCodeTag            (void)  :{virtual} int
```

Public Methods 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.

4. The cdevData Class

Overview of the cdevData Class

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

Figure 4: Object model of the cdevData class

class cdevData		
methods:		
tagC2I	**(char *, int *)**	**:int**
tagI2C	**(int, char * &)**	**:int**
insertTag	**(int, char *)**	**:void**
addTagCallback	**(cdevTagTableCallback*)**	**:void**
delTagCallback	**(cdevTagTableCallback*)**	**:void**
readTagTable	**(int *&,char **&,int &)**	**:int**
operator ==	**(cdevData & data)**	**:int**
operator !=	**(cdevData & data)**	**:int**
operator =	**(cdevData &)**	**:cdevData &**
operator char	**(void)**	**:char**
operator short	**(void)**	**:short**
operator unsigned short	**(void)**	**:unsigned short**
operator int	**(void)**	**:int**
operator unsigned int	**(void)**	**:unsigned int**
operator long	**(void)**	**:long**
operator unsigned long	**(void)**	**:unsigned long**
operator float	**(void)**	**:float**
operator double	**(void)**	**:double**
asciiDump	**(FILE *)**	**:void**
xdrSize	**(size_t *, size_t *)**	**:int**
xdrExport	**(char **, size_t *)**	**:int**
xdrExport	**(char *, size_t, size_t)**	**:int**
xdrImport	**(char *, size_t)**	**:int**
remove	**(...)**	**:void**
changeTag	**(...)**	**:int**
getType	**(...)**	**:cdevDataTypes**
getDim	**(...)**	**:int**
getElems	**(...)**	**:int**
getBounds	**(...)**	**:int**
setBounds	**(...)**	**:int**
insert	**(...)**	**:int**
get	**(...)**	**:int**
find	**(...)**	**:int**

Public Methods of the cdevData Class

tagC2I

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

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

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

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

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

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

addTagCallback *static void addTagCallback(cdevTagTableCallback * cb);*

This method is used to install a cdevTagTableCallback object in the global tag table that will be called each time a new tag is inserted into the global tag table. This method may be used by service developers to detect when new tags are created and update the corresponding tag mapping tables on remote servers.

The *cdevTagTableCallback* class contains a user defined *callback* method that takes as its parameters the new tag's integer identifier and its string equivalent.

delTagCallback *static void delTagCallback(cdevTagTableCallback * cb);*

This method removes the specified callback object from the global tag table. The callback should have been previously installed using the addTagCallback method.

readTagTable *static int readTagTable(int *&tag, char **&ctag, int &ntag);*

This method will read the complete contents of the global tag table and use the data to populate the caller provided variables. The *ntags* parameter will be set to the number of tags that were found in the global tag table.

The *tag* array and the *ctag* array will both be allocated to a size sufficient to hold the resultant data. The *tag* parameter will contain the list of integer tags and the *ctag* array will contain the list of corresponding character strings.

The caller will be responsible for deleting the arrays that are returned by this method. Because the *ctag* array contains pointers to the actual strings from the global tag table, the caller should delete the array only (not the individual elements).

Operator = *cdevData & operator = (cdevData & data);*

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

Cast operators

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

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

Operator ==

int operator == (cdevData & data);

This operator will do a deep comparison of two cdevData objects, including a comparison of the individual elements of array data. The order in which the tagged data items were inserted into either object is unimportant.

Operator !=

int operator != (cdevData & data);

This operator will do a deep comparison of two cdevData objects, including a comparison of the individual elements of array data. The order in which the tagged data items were inserted into either object is unimportant.

asciiDump

*void asciiDump (FILE * fp);*

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

xdrSize

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

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

xdrExport

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

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

xdrExport

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

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

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

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

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

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

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

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

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

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

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

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

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

getElems

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

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

getBounds

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

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

setBounds

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

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

insert (*scalar*)

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

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

insert (array)

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

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

insert (character string)

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

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

get

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

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

get (character string)

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

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

find

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

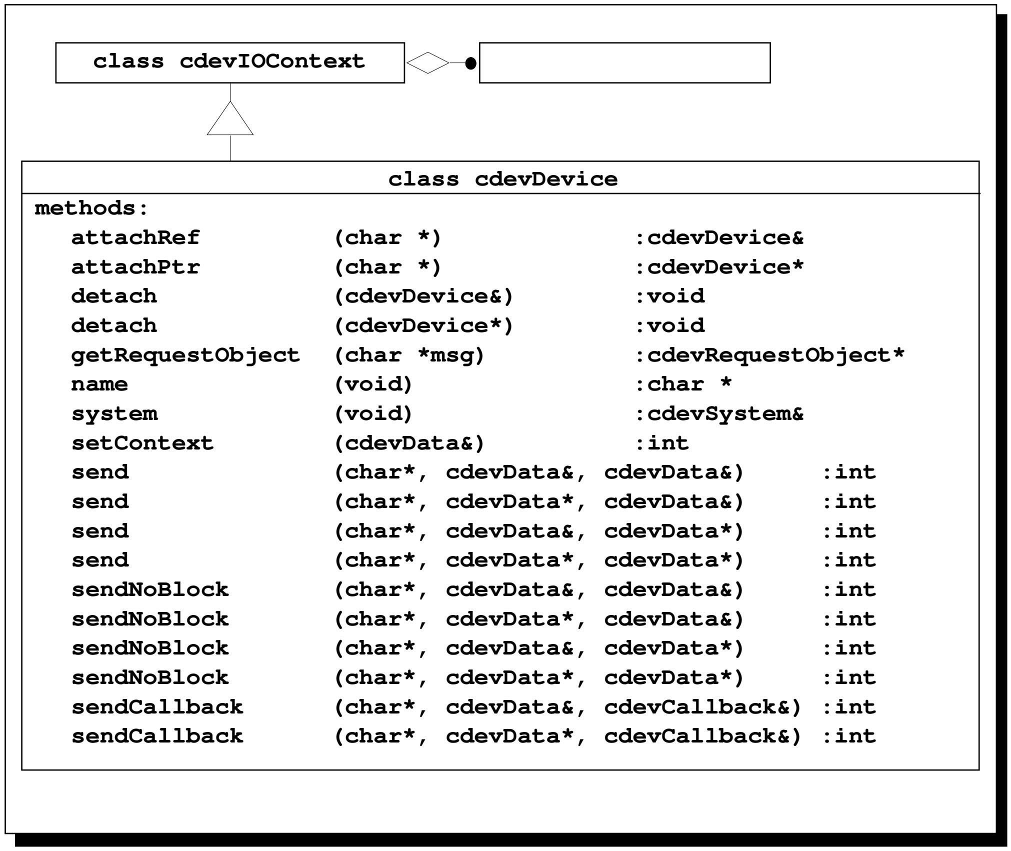
Retrieves a pointer to the data within the cdevData object. The caller is responsible for ensuring that the receiving pointer is of the correct data type for the data within the object. This function is overloaded to support both integer and character string tag identifiers.

5. The cdevDevice Class

Overview of the cdevDevice Class

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

Figure 5: Object model of the cdevDevice class



Public Methods of the cdevDevice Class

attachRef

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

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

attachPtr

static cdevDevice attachPtr (char *name);*

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

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

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

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

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

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

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

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

Returns the name of the device.

system *cdevSystem& system (void) const;*

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

setContext *int setContext (cdevData& cxt);*

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

getContext *cdevData & getContext (void);*

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

setPrivate *void setPrivate (void * data);*

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

getPrivate *void * getPrivate (void);*

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

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

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

Figure 6: Return codes generated by the send method.

CDEV_WARNING:	*Failure of the request in 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:	*Operation is not supported on device.*
CDEV_NOTCONNECTED:	*Not connected to low level network service.*
CDEV_IOFAILED:	*Low level network service IO failed.*
CDEV_CONFLICT:	*Conflicts of data types or tags.*
CDEV_NOTFOUND:	*Cannot find specified data in cdevData.*
CDEV_TIMEOUT:	*Time out.*
CDEV_CONVERT:	*cdevData conversion error.*
CDEV_OUTOFRANGE:	*Value out of range for device attribute.*
CDEV_NOACCESS:	*Insufficient access to perform request.*
CDEV_ACCESSCHANGED:	*Change in access permission of device.*
CDEV_DISCONNECTED:	*The service has lost contact with the device.*
CDEV_RECONNECTED:	*The service has regained contact with the device.*

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

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

sendCallback

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

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

6. The cdevError Class

Overview of the cdevError Class

The *cdevError* C++ class is an abstract base class that provides error handling services that are used by both the cdevService object and the cdevSystem object. It's constructors are protected to prevent it from being instantiated directly.

Figure 7: Object model of the cdevError class

```
                          class cdevError
methods:
   autoErrorOn      ( void )              {virtual} : int
   autoErrorOff     ( void )              {virtual} : int
   reportError      ( ... )               {virtual} : int
   setErrorHandler  (cdevErrorHandler )   {virtual} : void
   setThreshold     ( int )               {virtual} : void
   className       ( void )             {virtual} : char *
```

Public Methods of the cdevError Class

autoErrorOn

int autoErrorOn (void);

Informs the cdevError class 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 class.

autoErrorOff

int autoErrorOff (void);

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

reportError

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

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

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

CDEV_SEVERITY_INFO No error.

CDEV_SEVERITY_WARN	An error occurred that should not impact continued processing.
CDEV_SEVERITY_ERROR	An error occurred and should be corrected before continuing.
CDEV_SEVERITY_SEVERE	A severe or fatal error has occurred and cdev cannot continue normal execution.

setErrorHandler

void setErrorHandler (cdevErrorHandler handler);

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

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

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

setThreshold

void setThreshold (int errorThreshold);

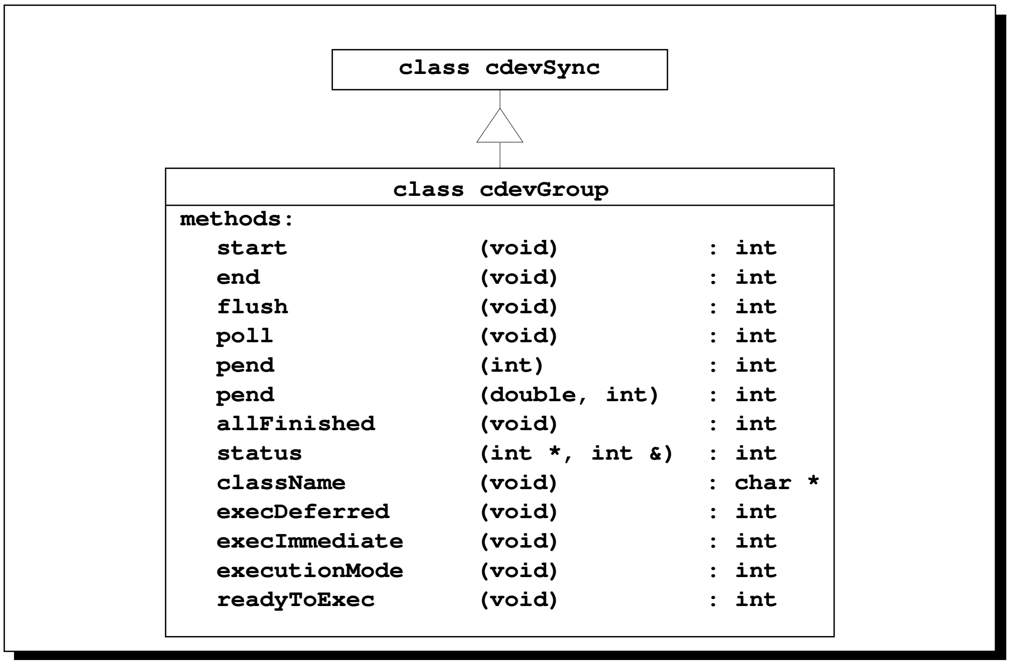
Used to specify the level of severity at which errors should be submitted to the error handler. The value of *errorThreshold* should be one of the severity levels specified in the *reportError* method.

7. The cdevGroup Class

Overview of the cdevGroup Class

The *cdevGroup* C++ class provides the mechanisms that cdev uses to collect together a list of operations to be executed, perhaps repeatedly, as a group, and to detect the completion of multiple requests that form a group.

Figure 8: Object model of the cdevGroup class



Public Methods of the cdevGroup Class

cdevGroup

cdevGroup (unsigned blockSize, cdevSystem& system);

This is the constructor for the cdevGroup object. It receives a blockSize parameter which specifies the maximum number of cdevTranObj objects that may be added and a cdevSystem parameter that the group object will be associated with. If no parameters are specified blockSize will be set to DEFAULT_BLOCK_SIZE and the system will be set to cdevSystem::defaultSystem(). The mode of the constructed group will be CDEV_EXEC_IMMEDIATE.

start

int start (void);

Starts a group of transactions. This means that each time a message is sent *asynchronously* to a device, the associated *cdevTranObj* will be included in this group. The system will stop adding transactions to this group after the *end* method has been executed.

If the mode of the group is CDEV_EXEC_IMMEDIATE, the operations are immediately forwarded to the service and to the appropriate server. If the mode of the group is

CDEV_EXEC_DEFERRED, the operation is kept in a list within the group, and sent to the service/server when flush() or pend() is called.

end

int end (void);

Informs the system that no more new transactions should be included in this group object.

flush

int flush (void);

Flushes any pending outbound requests to their respective services.

If the group is in deferred execution mode, and the grouped commands have either not yet been flushed, or have previously been flushed and have all completed, then flush will send the commands to the underlying services. After completion, flush may be called again to re-execute the grouped commands.

poll

int poll (void);

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

pend

int pend (int fd);

Automatically end and flush the group if it is started, and wait until all operations within the group are completed or until a timeout occurs.

In particular: 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 cdevGroup 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 cdevGroup object will call the respective cdevService to process the event and dispatch any asynchronous callbacks that are ready.

pend

int pend (double seconds, int fd);

Automatically end and flush the group if it is started, and wait until all operations within the group are completed or until a timeout occurs.

In particular, 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 cdevGroup 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 cdevGroup object will call the respective cdevService to process the event and dispatch any asynchronous callbacks that are ready.

allFinished *int allFinished (void);*

Checks the status of the cdevGroup object to determine if all transactions have been completed. Returns 1 if all transactions are complete or 0 if they are not.

status *int status (int status[], int &numTransactions);*

Retrieves the status of the individual transaction objects that are stored within the *cdevGroup* object. The cdevGroup class will populate the *status* array with up to *numTransactions*. The *numTransactions* parameter will then be set to the number of transactions that were actually loaded. If a *status* entry is 0, then the associated transaction has been completed. Returns CDEV_SUCCESS on successful completion, or CDEV_INVALIDARG if the *status* array is too small.

execDeferred *int execDeferred (void);*

execImmediate *int execImmediate (void);*

Changes the behaviour of execution of commands inside the group. execDeferred will buffer all commands until pend or poll flush is called; execImmediate will flush commands to underlying services immediately.

executionMode *int executionMode (void) const;*

Return execution mode: CDEV_EXEC_DEFERRED or CDEV_EXEC_IMMEDIATE.

readyToExec *int readyToExec (void) const;*

For deferred mode group only. Return execution stage:

0 = still in the buffered stage.

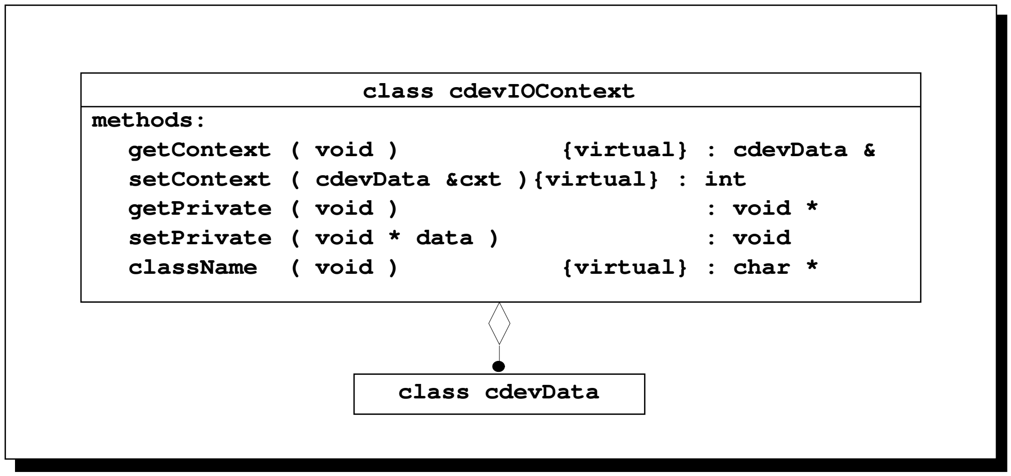
1 = ready to flush all buffered commands.

8. The cdevIOContext Class

Overview of the cdevIOContext Class

The *cdevIOContext* C++ class is an abstract base class that is inherited by both the *cdevRequestObject* class and the *cdevDevice* class. This class defines the mechanisms that are used to store and retrieve context information.

Figure 9: Object model of the cdevIOContext class



Public Methods of the cdevIOContext Class

getContext

cdevData & getContext (void);

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

setContext

int setContext (cdevData& cxt);

Used to insert a cdevData object containing tagged values that control optional behavior of the underlying device. The context is often used to specify which properties (value, status, severity) a device returns in response to a “get” message. .

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 object. The pointer can be retrieved later using the *getPrivate* method.

9. The cdevRequestObject Class

Overview of the cdevRequestObject Class

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

Figure 10: Object model of the cdevRequestObject class

Public Methods of the cdevRequestObject Class

attachRef

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

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

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

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

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

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

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

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

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

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

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

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

message *char *message (void) const;*

Retrieves the message string that is associated with this cdevRequestObject.

device *cdevDevice &device (void) const;*

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

system *cdevSystem& system (void) const;*

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

service *cdevService& service (void) const;*

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

getState *int getState (void);*

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

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

CDEV_STATE_INVALID:	Object is invalid.

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

getAccess

int getAccess (void);

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

CDEV_ACCESS_NONE:	No access to attribute.
CDEV_ACCESS_READONLY:	Read-only access.
CDEV_ACCESS_WRITE:	Read-write access.

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

setContext

int setContext (cdevData& cxt);

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

getContext

cdevData & getContext (void);

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

getPrivate

*void * getPrivate (void);*

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

setPrivate

*void setPrivate (void * data);*

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

send

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

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

Figure 11: Return codes generated by the send method.

```
CDEV_WARNING:          Failure of the function is non-consequential.
CDEV_SUCCESS:          The message was processed successfully.
CDEV_ERROR:            Failed to process message.
CDEV_INVALIDOBJ:       Invalid cdev request object used.
CDEV_INVALIDARG:       Invalid argument passed to cdev call.
CDEV_INVALIDSVC:       Wrong service during dynamic loading.
CDEV_NOTCONNECTED:     Not connected to low level network service.
CDEV_INVALIDOP:        The operation is not supported by the device.
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:     Service has lost contact with the device.
CDEV_RECONNECTED:      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.

10. The cdevSelector Class

Overview of the cdevSelector Class

The *cdevSelector* C++ class is a utility class that may be used by service developers. It allows a service that does not use network I/O to provide a locally managed file descriptor to the *cdevSystem* object. The service may then use the *insertEvent* method to signal the *cdevSystem* object that the services needs attention.

Figure 12: Object model of the cdevSelector class

```
                    class cdevSelector
methods:
   insertEvent        (int)      :  int
   removeEvent        (int)      :  int
   purge              (void)     :  void
   writefd            (void)     :  int
   readfd             (void)     :  int
```

Public Methods of the cdevSelector Class

insertEvent

int insertEvent (int numEvents);

Inserts the number of bytes specified in the numEvents parameter into the file descriptor. If numEvents is not specified, then one byte will be written.

removeEvent

int removeEvent (int numEvents);

Removes the number of bytes specified in the *numEvents* parameter from the file descriptor. If *numEvents* is not specified, then one byte will be read. This function uses the ioctl function to ensure that the *numEvents* parameter does not exceed the actual number of bytes available from the file descriptor.

purge

void purge (void);

Removes all bytes from the file descriptor.

writefd

int writefd (void);

Returns the write file descriptor that is stored within the *cdevSelector* object.

readfd

int readfd (void);

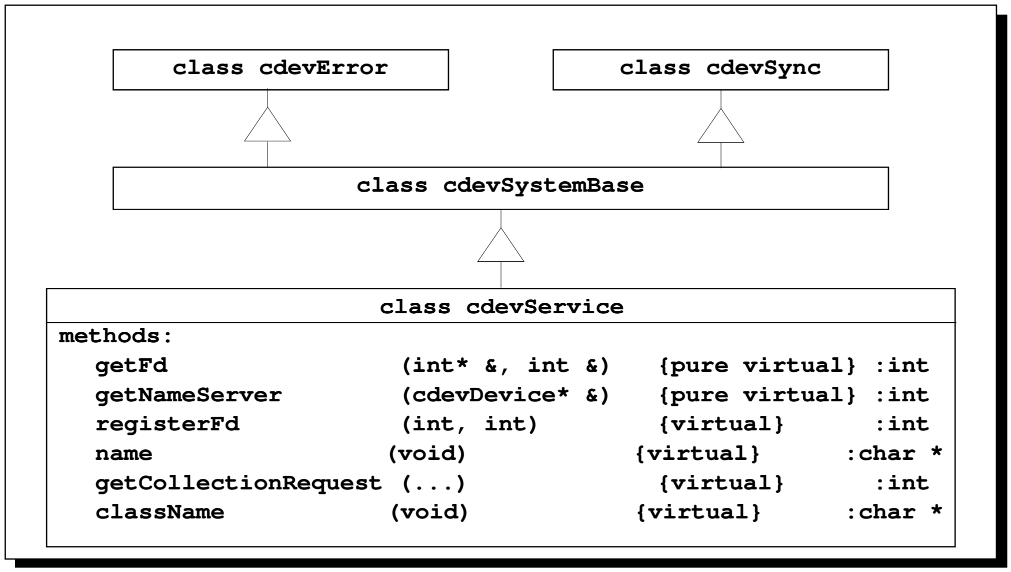
Returns the read file descriptor that is stored within the *cdevSelector* object.

11. 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 the 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.

Figure 13: Object model of the cdevService class



Public Methods of the cdevService Class

flush

int flush (void);

Flushes any pending outbound requests to their respective devices. 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 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”.

12. The cdevSync Class

Overview of the cdevSync Class

The *cdevSync* C++ class is a pure virtual base class that declares the I/O operations that must be supported by the cdevService object and the cdevSystem object. Because several of its functions are pure virtual, it cannot be instantiated directly.

Figure 14: Object model of the cdevSync class

```
                    class cdevSync
methods:
   flush      ( void )          {pure virtual}   :int
   poll       ( void )          {pure virtual}   :int
   pend       ( int )           {pure virtual}   :int
   pend       ( double, int )   {pure virtual}   :int
   className  ( void )          {virtual}        :char *
```

Public Methods of the cdevSync Class

flush

int flush (void);

Flushes any pending outbound requests to their respective services. This is a pure virtual method that is implemented in derived classes.

poll

int poll (void);

Directly polls each of the underlying services for activity, and delivers any asynchronous callbacks that are ready. This is a pure virtual function that is implemented in the derived classes.

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 cdevSync 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 cdevSync 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 is implemented in the derived classes.

pend

int pend (double seconds, int fd);

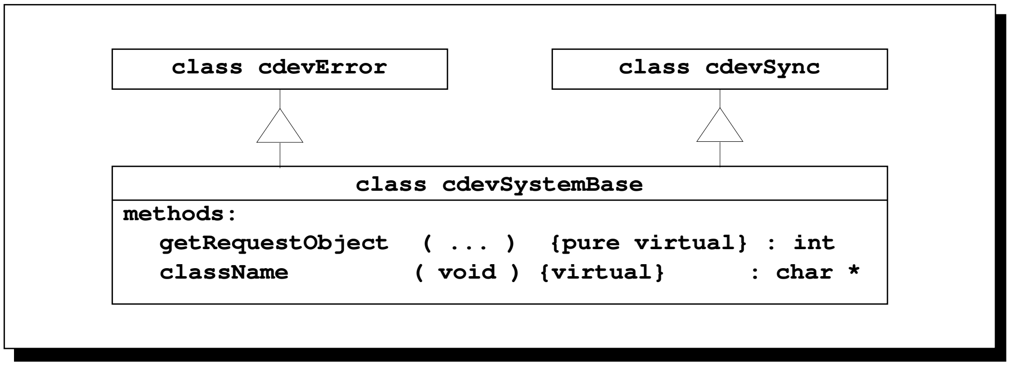
Processes all I/O events that occur on the file descriptor during the specified period of time. If the fd parameter is not provided, the cdevSync object will wait for I/O events on all of its contained file descriptors. When an event occurs on one of the file descriptors, the cdevSync object will call the appropriate functions to process the event and dispatch any asynchronous callbacks that are ready. This is a pure virtual function that is implemented in the derived classes.

13. The cdevSystemBase Class

Overview of the cdevSystemBase Class

The *cdevSystemBase* C++ class is a pure virtual base class that inherits the capabilities of both the cdevError class and the cdevSync class. It then declares the functions necessary to obtain a request object. Because several of its functions are pure virtual, it cannot be instantiated directly.

Figure 15: Object model of the cdevSystemBase class



Public Methods of the cdevSystemBase Class

getRequestObject

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

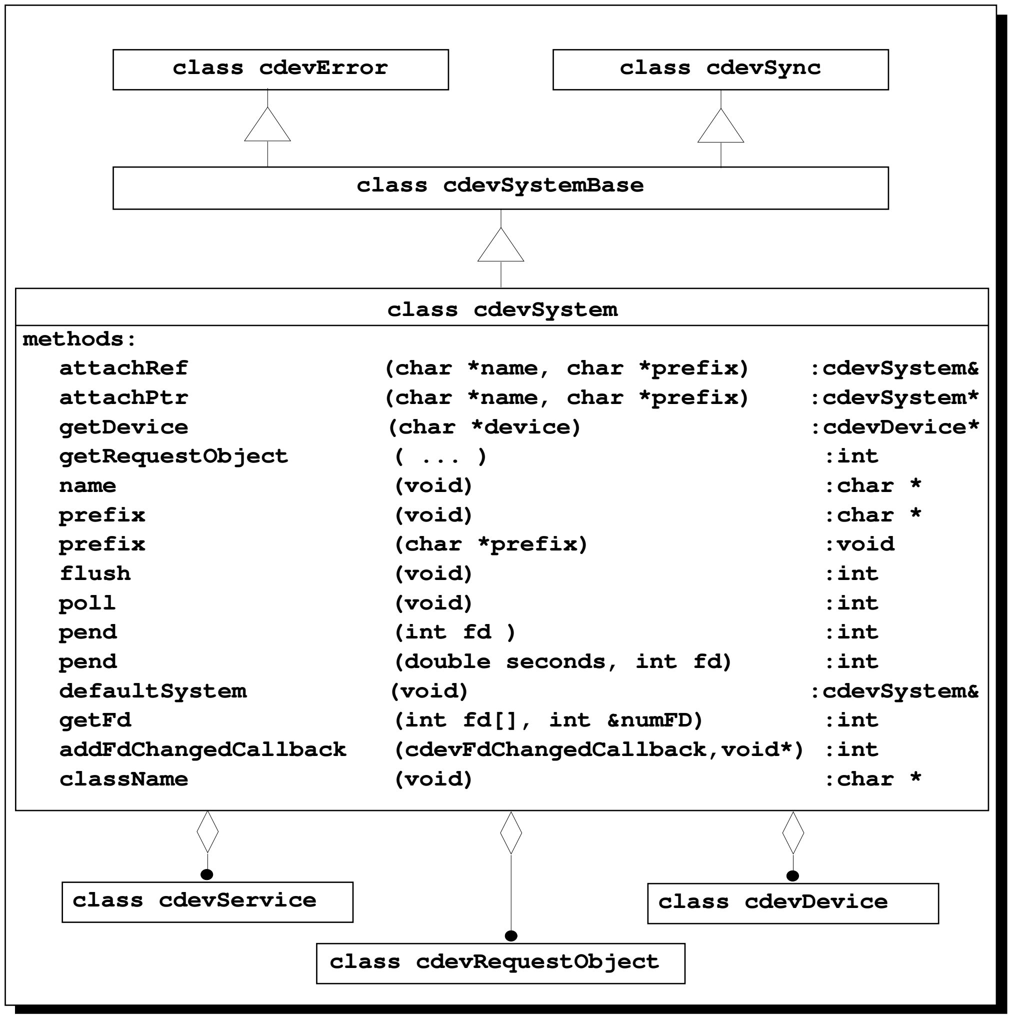
Obtains a reference to a cdevRequestObject from a specified device name and message name. The object that is produced by this function will be owned and controlled by the cdevSystem object that created it. This means that *flush, poll* and *pend* commands must be routed through the owning cdevSystem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

14. The cdevSystem Class

Overview of the cdevSystem Class

The *cdevSystem* C++ class provides the mechanisms that cdev uses to manage and communicate with multiple services. This class also provides the caller with the tools necessary to instantiate *cdevDevice* objects and *cdevRequestObject* objects.

Figure 16: Object model of the cdevSystem class



Public Methods of the cdevSystem Class

attachRef

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

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

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

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

defaultSystem *static cdevSystem &defaultSystem (void);*

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

getDevice *cdevDevice* getDevice (char *device);*

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

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

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

name *char *name (void);*

Returns the name of this cdevSystem object.

prefix *char *prefix (void);*

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

prefix *void prefix (char *pre);*

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

flush *int flush (void);*

Flushes any pending outbound requests to their respective services.

poll *int poll (void);*

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

pend *int pend (int fd);*

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

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

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

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

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

addFdChangedCallback

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

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

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

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

autoErrorOn *int autoErrorOn (void);*

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

autoErrorOff *int autoErrorOff (void);*

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

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

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

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

CDEV_SEVERITY_INFO:	No error.
CDEV_SEVERITY_WARN:	An error occurred that should not impact continued processing.
CDEV_SEVERITY_ERROR:	An error occurred and should be corrected before continuing.
CDEV_SEVERITY_SEVERE:	A severe or fatal error has occurred and cdev cannot continue normal execution.

setErrorHandler *void setErrorHandler (cdevErrorHandler handler);*

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

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

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

setThreshold *void setThreshold (int errorThreshold);*

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

15. 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 may also be used to place a transaction into a group of requests using a 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.

Figure 17: Object model of the cdevTranObj class

```
                class cdevTranObj
properties:
   system_         : cdevSystem *
   reqObj_         : cdevRequestObject *
   resultData_     : cdevData *
   userCallback_   : cdevCallback *

methods:
   status             (void) : int
   removeFromGrps     (void) : int
   enableDeleteCbk    (void) : void
   disableDeleteCbk   (void) : void
```

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.

16. Language Specification for the cdev DDL file

Language Specification

In the following, required language elements are in double quotes, optional elements are in square brackets. Names are arbitrary character strings. value is of any type. Plus sign indicates concatenation, vertical bar alternate choices.

Figure 18: Language specification for the cdev DDL file

```
specification = serviceSpecs + classSpecs + instances + aliases +
collections

serviceSpecs = serviceSpec + [serviceSpecs]
serviceSpec = "service" + name + "{" + "tags" + "{" + tagList + "}"
+ "}"
tagList = tag + ["," + tagList]

classSpecs = classSpec + [classSpecs]
classSpec = "class" + name + [" : " + parentList] + "{" +
         verbs + attributes + messages
parentList = name + ["," + parentList]
verbs = "verbs" + "{" + verbList + "}"
verbList = name + ["," + verbList]
attributes = "attributes" + "{" + attrList + "}"
attrList = name + serviceName + ["{" + serviceData + "}"] + [";" +
attrList]
serviceData = name + "=" + value
messages = "messages" + "{" + msgList + "}"
msgList = msg + serviceName + ["{" + serviceData + "}"] + [";" +
msgList]
msg = name | """string"""

instances = className + ":" instanceList + ";"
instanceList = instance + [" " + instanceList]

aliases = alias + [" " + aliases]
alias = "alias " + aliasname + " " + actualname

collections = collection + " " + name + " : " instanceList + ";"
instanceList = instance + [" " + instanceList]
```

17. Regular Expressions

Overview of Regular Expressions

A regular expression is a mechanism supported by many utilities for locating and manipulating patterns in text. Basic regular expression (RE) notation and construction rules apply to utilities defined as using basic REs. Any exceptions to the following rules are noted in the descriptions of the specific utilities that use REs.

Ordinary Characters

An ordinary character is an RE that matches itself. An ordinary character is any character in the supported character set except <newline> and the regular expression special characters listed in Special Characters below. An ordinary character preceded by a backslash (\) is treated as the ordinary character itself, except when the character is (,), {, or }, or the digits 1 through 9 (see REs Matching Multiple Characters). Matching is based on the bit pattern used for encoding the character; not on the graphic representation of the character.

Special Characters

A regular expression special character preceded by a backslash is a regular expression that matches the special character itself. When not preceded by a backslash, such characters have special meaning in the specification of REs. Regular expression special characters and the contexts in which they have special meaning are:

. [\
The period, left square bracket, and backslash are special except when used in a bracket expression (see RE Bracket Expression).

*****
The asterisk is special except when used in a bracket expression, as the first character of a regular expression, or as the first character following the character pair \((see REs Matching Multiple Characters).

^
The circumflex is special when used as the first character of an entire RE (see Expression Anchoring) or as the first character of a bracket expression.

$
The dollar sign is special when used as the last character of an entire RE (see Expression Anchoring).

delimiter
Any character used to bound (i.e., delimit) an entire RE is special for that RE.

period
A period (.), when used outside of a bracket expression, is an RE that matches any printable or nonprintable character except <newline>.

Bracket Expression

A bracket expression enclosed in square brackets ([ ]) is an RE that matches a single collating element contained in the nonempty set of collating elements represented by the bracket expression. The following rules apply to bracket expressions:

bracket expression A bracket expression is either a matching list expression or a non-matching list expression, and consists of one or more expressions in any order. Expressions can be: collating elements, collating symbols, noncollating characters, equivalence classes, range expressions, or character classes. The right bracket (]) loses its special meaning and represents itself in a bracket expression if it occurs first in the list (after an initial ^, if any). Otherwise, it terminates the bracket expression (unless it is the ending right bracket for a valid collating symbol, equivalence class, or character class, or it is the collating element within a collating symbol or equivalence class expression). The special characters

. * [\ (period, asterisk, left bracket, and backslash) lose their special meaning within a bracket expression.

matching list A matching list expression specifies a list that matches any one of the characters represented in the list. The first character in the list cannot be the circumflex. For example, [abc] is an RE that matches any of a, b, or c.

non-matching list A non-matching list expression begins with a circumflex (^), and specifies a list that matches any character except <newline> and the characters represented in the list. For example, [^abc] is an RE that matches any character except <newline> or a, b, or c. The circumflex has this special meaning only when it occurs first in the list, immediately following the left square bracket.

collating element A collating element is a sequence of one or more characters that represents a single element in the collating sequence as identified via the most current setting of the locale category LC_COLLATE (see setlocale(3C)).

collating symbol A collating symbol is a collating element enclosed within bracket-period ([.....]) delimiters. Multi-character collating elements must be represented as collating symbols to distinguish them from single-character collating elements. For example, if the string ch is a valid collating element, then [.ch.] is treated as an element matching the same string of characters, while ch is treated as a simple list of the characters c and h. If the string within the bracket-period delimiters is not a valid collating element in the current collating sequence definition, the symbol is treated as an invalid expression.

noncollating char A noncollating character is a character that is ignored for collating purposes. By definition, such characters cannot participate in equivalence classes or range expressions.

equivalence class An equivalence class expression represents the set of collating elements belonging to an equivalence class. It is expressed by enclosing any one of the collating elements in the equivalence class within bracket-equal ([=...=]) delimiters. For example, if ,,and A belong to the same equivalence class, then [[=a=]b], =]b], and [[=A=]b] are each equivalent toAb].

range expression

A range expression represents the set of collating elements that fall between two elements in the current collation sequence as defined via the most current setting of the locale category LC_COLLATE (see setlocale(3C)). It is expressed as the starting point and the ending point separated by a hyphen (-).

The starting range point and the ending range point must be a collating element, collating symbol, or equivalence class expression. An equivalence class expression used as an end point of a range expression is interpreted such that all collating elements within the equivalence class are included in the range. For example, if the collating order is A, a, B, b, C, c, ch, D, d; and A and a constitute an equivalence class, then the expression [[=a=]-D] is treated as [AaBbCc[.ch.]D].

Both starting and ending range points must be valid collating elements, collating symbols, or equivalence class expressions, and the ending range point must collate equal to or higher than the starting range point; otherwise the expression is invalid. For example, with the above collating order and assuming that E is a noncollating character, then both the expressions [[=A=]-E] and [d-a] are invalid.

An ending range point can also be the starting range point in a subsequent range expression. Each such range expression is evaluated separately. For example, the bracket expression [a-m-o] is treated as [a-mm-o].

The hyphen character is treated as itself if it occurs first (after an initial ^, if any) or last in the list, or as the rightmost symbol in a range expression. As examples, the expressions [-ac] and [ac-] are equivalent and match any of the characters a, c, or -; the expressions [^-ac] and [^ac-] are equivalent and match any characters except <newline>, a, c, or -; the expression [%--] matches any of the characters in the defined collating sequence between % and - inclusive; the expression [--@] matches any of the characters in the defined collating sequence between - and @ inclusive; and the expression [a--@] is invalid, assuming - precedes a in the collating sequence.

character class

A character class expression represents the set of characters belonging to a character class, as defined via the most current setting of the locale category LC_CTYPE. It is expressed as a character class name enclosed within bracket-colon ([: :]) delimiters.

Valid character class expressions and the class they represent are:

[:alpha:] letters

[:upper:] upper-case letters

[:lower:] lower-case letters

[:digit:] decimal digits

[:xdigit:] hexadecimal digits

[:alnum:] letters or decimal digits

[:space:] characters producing white- space in displayed text

[:print:] printing characters

[:punct:] punctuation characters

[:graph:] characters with a visible representation

[:cntrl:] control characters

Matching Multiple Characters

The following rules may be used to construct REs matching multiple characters from REs matching a single character:

RERE

The concatenation of REs is an RE that matches the first encountered concatenation of the strings matched by each component of the RE. For example, the RE bc matches the second and third characters of the string abcdefabcdef.

RE*

An RE matching a single character followed by an asterisk (*) is an RE that matches zero or more occurrences of the RE preceding the asterisk. The first encountered string that permits a match is chosen, and the matched string will encompass the maximum number of characters permitted by the RE. For example, in the string abbbcdeabbbbbbcde, both the RE b*c and the RE bbb*c are matched by the

substring bbbc in the second through fifth positions. An asterisk as the first character of an RE loses this special meaning and is treated as itself.

\(RE\)

A subexpression can be defined within an RE by enclosing it between the character pairs \(and \). Such a subexpression matches whatever it would have matched without the \(and \). Subexpressions can be arbitrarily nested. An asterisk immediately following the \(loses its special meaning and is treated as itself. An asterisk immediately following the \) is treated as an invalid character.

\n

The expression \n matches the same string of characters as was matched by a subexpression enclosed between \(and \) preceding the \n. The character n must be a digit from 1 through 9, specifying the n-th subexpression (the one that begins with the n-th \(and ends with the corresponding paired \). For example, the expression ^\(.*\)\1$ matches a line consisting of two adjacent appearances of the same string.

If the \n is followed by an asterisk, it matches zero or more occurrences of the subexpression referred to. For example, the expression \(ab\(cd\)ef\)Z\2*Z\1 matches the string abcdefZcdcdZabcdef.

RE\{m,n\}

An RE matching a single character followed by \{m\}, \{m,\}, or \{m,n\} is an RE that matches repeated occurrences of the RE. The values of m and n must be decimal integers in the range 0 through 255, with m specifying the exact or minimum number of occurrences and n specifying the maximum number of occurrences. \{m\} matches exactly m occurrences of the preceding RE, \{m,\} matches at least m occurrences, and \{m,n\} matches any number of occurrences between m and n, inclusive.

The first encountered string that matches the expression is chosen; it will contain as many occurrences of the RE as possible. For example, in the string abbbbbbbc the RE b\{3\} is matched by characters two through four, the RE b\{3,\} is matched by characters two through eight, and the RE b\{3,5\}c is matched by characters four through nine.

Expression Anchoring

An RE can be limited to matching strings that begin or end a line (i.e., anchored) according to the following rules:

- A circumflex (^)as the first RE anchors the expression to the beginning of a line; only strings starting at the first character of a line are matched by the RE. For example, the RE ^ab matches the string ab in the line abcdef, but not the same string in the line cdefab.

- A dollar sign ($) as the last character of an RE anchors the expression to the end of a line; only strings ending at the last character of a line are matched by the RE. For example, the RE ab$ matches the string ab in the line cdefab, but not the same string in the line abcdef.

- An RE anchored by both ^ and $ matches only strings that are lines. For example, the RE ^abcdef$ matches only lines consisting of the string abcdef.